UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE PRODUÇÃO

PROBLEMA DO CAIXEIRO VIAJANTE

UM ALGORITMO PARA RESOLUÇÃO DE PROBLEMAS DE GRANDE PORTE BASEADO EM BUSCA LOCAL DIRIGIDA

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA, PARA OBTENÇÃO DO GRAU DE "MESTRE EM ENGENHARIA".

MARCO ANTONIO PEREIRA RODRIGUES

FLORIANÓPOLIS
SANTA CATARINA – BRASIL
2000

PROBLEMA DO CAIXEIRO VIAJANTE

UM ALGORITMO PARA RESOLUÇÃO DE PROBLEMAS DE GRANDE PORTE BASEADO EM BUSCA LOCAL DIRIGIDA

MARCO ANTONIO PEREIRA RODRIGUES

Esta dissertação foi julgada adequada para a obtenção do Título de:

"**MESTRE EM ENGENHARIA**"

Especializada em Engenharia da Produção e aprovada em sua forma final pelo Programa de Pós-Graduação.

_____

Prof. Ricardo Miranda Barcia, Ph.D.
Coordenador

BANCA EXAMINADORA:

_____

Prof. Sérgio Fernando Mayerle, Dr.
Orientador

_____

Prof. Antônio Sérgio Coelho, Dr.

_____

Prof. Aran Bey Tcholakian Morales, Dr.

_____

Prof.ª Mirian Gonçalves Buss, Dra.

Dedico a
João Bosco e Marildes,
meus pais; Maria Alice,
minha esposa; e Gabriel e
Ana Beatriz, meus filhos.

AGRADECIMENTOS

A Deus, por ter me dado a força necessária para concluir este trabalho.

Ao Professor Dr. Sérgio Fernando Mayerle, pelo modo interessado, participativo e eficiente, com que me orientou, muitíssimo obrigado.

Aos amigos que acreditaram no sonho e na causa Alcinéia Motta, Edijane Paredes, Elianete Ferreira Lima, Francy Galvão, Iracema Reis, Ismar dos Santos, Jaqueline Rafael, Jerusa Cipriano, José Feitosa, Marcos de Sousa, Maria de Mendonça, Maria Aparecida. Silva, Maria de Fátima Rodrigues, Maria Eliny, Maria Vieira, Marli Pereira, Nilza Maria de Souza, Patrícia Cabral e Silva, Paulo Arruda, Paula Fróes, Regina Chaves, Roberto Alves, Rodrigo de Lima, Sebastião Ayres, Shirley, Silverlane de Oliveira, Socorro Moraes, Valdereis, Vera Lúcia Farias e Waldemarina de Souza, muitíssimo obrigado. Nós vencemos!

Aos amigos João C. do Lago Neto, Nilomar Oliveira, Nilson Barreiros, Genoveva Azevedo e Jasyilene Abreu, que foram minha família em Florianópolis, meu muito obrigado.

Ao amigo Alcemir Ramos de Oliveira Filho, pela colaboração durante a fase inicial do curso, muito obrigado.

A Marco Antonio Cabral e Ana Cláudia P. Rodrigues, por terem facilitado a redução da distância entre mim e meus familiares, muito obrigado.

Enfim, a todos que acreditaram, torceram ou de alguma forma contribuíram para a realização deste trabalho, meu sincero agradecimento.

RESUMO

Neste trabalho é proposto um algoritmo para a resolução do Problema do Caixeiro Viajante (PCV), baseado em estratégia de particionamento, que atua em conjunto com a recém a apresentada metaheurística Busca Local Dirigida (BLD). Testes são realizados para avaliar a qualidade desse algoritmo, frente a um outro procedimento, também baseado em estratégia de particionamento, sobre problemas da biblioteca TSPLIB de Reinelt. Verificou-se que o algoritmo proposto é capaz de gerar bons resultados, em tempo relativamente curto. Algumas sugestões e considerações são apresentadas para o desenvolvimento de futuros trabalhos.

ABSTRACT

In this work, an algorithm is proposed for the resolution of Traveling Salesman Problem (TSP), based on clustering strategy, that acts together with the recently presented metaheuristic it Guided Local Search (GLS). Verification tests are accomplished to evaluate the quality of that algorithm, front to another procedure, also based on clustering strategy, in problems of the TSPLIB library of Reinelt. It was verified that the proposed algorithm, it is capable to generate good results, in time relatively short. Some suggestions and considerations are presented for the development of future works.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo II

2. Introdução

2.1. Considerações iniciais

O problema mais intensivamente estudado em otimização combinatorial é conhecido como o Problema do Caixeiro Viajante (PCV). Esse problema consiste na determinação da rota de menor custo para um vendedor que deseja visitar um conjunto finito de cidades. Para tanto, ele deverá iniciar a viagem em uma cidade qualquer, passar por todas as demais cidades exatamente uma vez, e então retornar para a cidade onde a rota teve início.
A origem desse problema não é precisamente estabelecida. Na década de 20, Karl Menger o divulgou no meio acadêmico de Viena. Na década de 30, já com o nome de PCV, Merril Flood o fez entre os matemáticos de Princeton. Finalmente, na década de 40, Flood leva o problema para seus colegas na *RAND Corporation*, onde rapidamente, ganha notoriedade em meio à comunidade de pesquisa operacional.
Apesar de sua descrição ser bastante simples, obter boas soluções não é tarefa fácil. Esse é, provavelmente, um fator que tem despertado em muitos o interesse em propor procedimentos para resolvê-lo. Tal interesse pode ser constatado através das centenas de textos publicados sobre o assunto.
A despeito da "mística" existente em torno do problema, muitas aplicações de interesse prático têm surgido, como: roteamento de veículos, corte de papel de parede, sequenciamento de tarefas e fabricação de chips de computador entre outras.

2.2. Objetivos

Os dois principais objetivos deste trabalho são:

a) verificar as potencialidades da recém apresentada metaheurística *Busca Local Dirigida* e sua utilização efetiva na resolução do *Problema do Caixeiro Viajante* (PCV);
b) apresentar e avaliar as potencialidades de um algoritmo, baseado em estratégias de particionamento e que atue em conjunto com essa nova metaheurística, na resolução de problemas com milhares de cidades, em um ambiente com limitação de recursos computacionais.

2.3. Motivação

À primeira vista, determinar a rota mais econômica para um caixeiro viajante parece ter pouco valor, pois é certo que não existem tantos caixeiros viajantes em busca de um algoritmo para isso. Entretanto, esse modelo é componente central na determinação de soluções de outros problemas de interesse prático. Encontrar as rotas mais econômicas para uma frota de veículos de distribuição é um exemplo típico. Neste tipo de problema, conhecido como o problema do roteamento de veículos, as rotas devem ser determinadas considerando não apenas o tamanho do caminho a ser percorrido, mas também restrições associadas a demandas existentes em cada ponto de entrega, limitações de tempo, diferenças de capacidade dos veículos (de carga ou autonomia), entre outras.
O problema de roteamento de veículos é uma extensão natural do PCV. Existem outras aplicações, aparentemente menos relacionadas, mas que também fazem uso desse mesmo modelo. Eis alguns exemplos:

- *Sequenciamento de tarefas*: suponha que *n* tarefas devem ser realizadas por uma máquina, e que entre a execução de duas tarefas consecutivas é necessária a execução de um reajuste na máquina. Existe um custo associado a cada reajuste. Seja c_{ij} o custo para realizar a tarefa *j* após a tarefa *i*. O problema

está em determinar em qual seqüência as tarefas devem ser executadas de modo a minimizar o somatório desses custos.

- *Perfuração de placas de circuitos impressos*: A fim de conectar as camadas de uma placa de circuitos e permitir a anexação de componentes, devem ser perfurados buracos de diâmetros diferentes na placa. Diferentes brocas devem ser utilizadas de acordo com o diâmetro do furo. Existe um custo (medido em unidades de tempo, por exemplo) associado à troca de uma broca por outra. Para tornar mínima a soma desses custos, os furos de um mesmo diâmetro devem ser realizados todos de uma só vez. Além disso, um tempo adicional é consumido no reposicionamento da broca entre um furo e outro. Supondo que esse tempo é proporcional à distância entre dois furos sucessivos, esse problema pode então ser modelado como uma série de PCV's, onde as cidades são as posições dos buracos a serem perfurados e a distância é o tempo necessário para a movimentação entre os furos. Em cada um dos PCV's o custo da operação é dado pelo somatório dos tempos de deslocamento da broca segundo a ordem programada de perfurações.
- *Fabricação de chips* – Um processo de fabricação de chips pode ser visto simplificadamente do seguinte modo: considere uma "bolacha" (placa redonda e fina de um cristal de silício sobre a qual centenas de circuitos integrados individuais são construídos antes de serem cortados em chips individuais), na qual serão gravadas uma sucessão de linhas (que correspondem às conexões elétricas entre diferentes componentes do chip). A máquina que faz a gravação, primeiro se move para a posição onde a linha começa, grava a linha particular (como especificado no projeto) a qual leva a um ponto diferente da "bolacha". A máquina então se move para o início de uma nova linha e assim sucessivamente. As linhas podem ser gravadas em qualquer ordem, mas para cada linha a gravação deve se dar na direção especificada (de um ponto de partida a um de término, estabelecidos previamente). A máquina deve começar em uma posição específica da bolacha para não interferir em uma outra que está sendo retirada da posição de gravação. O problema está na determinação da ordem com que as linhas serão gravadas de modo a consumir o menor tempo possível.
- *Raios-X em cristalografia* – Em cristalografia, alguns experimentos consistem em tomar um grande número de medidas da intensidade de raios-X sobre cristais, por meio de um detetor. A cada medição, é necessário que uma amostra do cristal seja colocada sobre um aparato e o medidor ajustado. A seqüência na qual as medidas devem ser feitas é determinada pela resolução de um PCV [Bla89].

Algumas outras aplicações citadas na literatura são: o corte de papel de parede por Garfinkel [Garf77] e o desenho de jogos de dardos por Eiselt e Laporte [Eis91].
De acordo com suas características, um PCV pode ser classificado como: simétrico, se as distâncias de ida e volta entre as cidades são iguais para todo par de cidades; caso contrário é dito assimétrico. Para pontos dispersos em um plano e considerada a distância euclidiana entre os pontos é chamado PCV euclidiano; se as distâncias entre as cidades são arbitrárias (aleatórias) é chamado de PCV com matriz de distância aleatória.
Ao longo deste trabalho, sempre que uma referência for feita ao PCV, deve-se ter em mente o caso simétrico, salvo menção em contrário.

2.4. Estrutura do trabalho

O presente trabalho está dividido em 5 capítulos. O primeiro é introdutório. No Capítulo II é exibida uma revisão dos procedimentos mais freqüentemente utilizados para resolver o PCV. No Capítulo III é apresentada a metaheurística *Busca Local Dirigida* e sua aplicação ao PCV. Nesse mesmo capítulo, é feita a descrição de um mecanismo de ajuste automático dos parâmetros desta metaheurística. No Capítulo IV é apresentado um algoritmo para resolução de problemas de grande porte, juntamente com os resultados da aplicação desse algoritmo a um conjunto de problemas da TSPLIB de Reinelt [Rei91]. No Capítulo V são feitas considerações sobre os resultados obtidos e recomendações para trabalhos futuros.

Capítulo III

2. Revisão da Literatura

2.1. Considerações iniciais

Neste capítulo serão apresentados os conceitos básicos sobre complexidade, otimização combinatorial e algumas das técnicas utilizadas na resolução do problema do caixeiro viajante.

2.2. Complexidade e otimização combinatorial.

Problemas de otimização combinatorial são encontrados em diversas situações. Elementos comuns a essa classe são os problemas de alocação, roteamento e programação de horários. Nestes problemas, o objetivo é assinalar valores a um conjunto de variáveis de decisão, de tal modo que uma função dessas variáveis (função objetivo) seja minimizada (ou maximizada) na presença de um conjunto de restrições. Formalmente, um problema de otimização combinatorial[1] é definido através de um conjunto finito $N=\{1,...,n\}$, com pesos c_j associados a cada $j \in N$, e um conjunto F formado por subconjuntos viáveis de N. Deseja-se determinar elementos de F, tais que o somatório dos pesos associados sejam mínimos, isto é, determinar

$$\min_{S \subseteq N} \{ \sum_{j \in S} c_j : S \in F \}.$$

Em problemas deste tipo, uma estratégia trivial para obtenção de soluções ótimas consiste na avaliação de todas as soluções viáveis e na escolha daquela que minimize o somatório dos pesos. O único inconveniente dessa estratégia está na chamada explosão combinatorial. Tomando como exemplo, um PCV com n cidades conectadas par a par, o número de soluções viáveis é da ordem $(n-1)!/2$. Pode-se ver na Tabela II-1 o número de possibilidades para alguns valores de n, juntamente com o tempo estimado para se resolver o problema usando essa estratégia, supondo que se possa avaliar 10^{12} soluções por segundo.

n	*(n-1)!/2*	*Em dias*	*Em anos*	*Em bilhões de anos*
10	$1{,}81\times10^{5}$	$2{,}10\times10^{-13}$	$5{,}75\times10^{-16}$	$5{,}75\times10^{-23}$
100	$4{,}67\times10^{155}$	$5{,}40\times10^{137}$	$1{,}48\times10^{135}$	$1{,}48\times10^{128}$
1000	$2{,}01\times10^{2564}$	$2{,}33\times10^{2546}$	$6{,}38\times10^{2543}$	$6{,}38\times10^{2536}$

Tabela III-1 : Tempo estimado

Como se vê, o tempo necessário para utilização dessa abordagem é proibitivo mesmo para valores pequenos de n. Assim, mecanismos mais inteligentes devem ser empregados na resolução deste tipo de problema.
O PCV não é um problema de fácil resolução. A razão disso não está no fato de existirem muitas soluções possíveis. Com efeito, um outro problema bem conhecido é a determinação da árvore de custo mínimo em um grafo completamente conectado. Neste caso, o número de árvores possíveis é muitas vezes maior que o número de rotas que satisfazem as restrições do PCV. Contudo, existem maneiras eficientes de encontrar uma solução de custo mínimo para o caso da árvore.
Na *Teoria da Complexidade* (*TC*) são fornecidos critérios para avaliação da dificuldade de resolução do PCV e de outros problemas.

[1] Neste texto será definido e tratado como um problema de minimização. A definição pode ser feita também em termos de maximização.

Inicialmente, em *TC*, um *problema* é definido como uma questão geral para a qual deve ser dada uma resposta, podendo tal questão ter muitas variáveis (ou parâmetros), cujos valores estão em aberto. Uma *instância de um problema é* obtida através da fixação desses valores e da especificação de quais propriedades uma solução para o problema deve possuir.

Formalmente, definem-se problemas através de um *esquema de representação,* formado por palavras de 1's e 0's que representam as instâncias e as soluções do problema. Com efeito, denomina-se problema a um subconjunto P de $\{0,1\}^* \times \{0,1\}^*$, onde $\{0,1\}^*$ denota o conjunto de todas as palavras finitas de 0's e 1's. Cada palavra $s \in \{0,1\}^*$ é chamada de *instância* ou *entrada* de P e cada $t \in \{0,1\}^*$ tal que $(s,t) \in P$ é chamada *solução* ou *saída* de P. Assume-se que para cada entrada em P existe pelo menos uma solução.
Para um esquema de representação específico, define-se *tamanho de uma entrada (L)* para uma *instância* de um problema, como o número de elementos que compõem a palavra que representa tal entrada.
Um *algoritmo é* uma seqüência de passos que devem ser executados para a obtenção de uma solução para um problema. Naturalmente, algoritmos diferentes podem ser propostos para resolver um mesmo problema.
Para um esquema de representação específico de um problema, a *função de complexidade de tempo* $f : N \to N$ de um algoritmo, expressa o máximo de tempo (operações elementares) necessário para resolver qualquer instância de tamanho $n \in N$.
Um algoritmo é denominado *algoritmo de tempo polinomial,* se sua função de complexidade de tempo *f* é tal que $f(n) \leq p(n)$ para todo $n \in N$, para algum polinômio *p*.
Existe uma classe de problemas chamados *problemas de decisão*. Tais problemas possuem apenas duas soluções possíveis (ou saídas), quais sejam "sim" ou "não". A classe formada por todos os problemas de decisão que possuem um *algoritmo de tempo polinomial* é denominada *classe P*.
Uma outra classe de problemas de decisão é a chamada *classe NP*. Esta classe é formada por todos os problemas de decisão com a seguinte propriedade: "Se a resposta para uma instância de P é "sim", então este fato pode ser provado em tempo polinomial".
Vê-se que $P \subseteq NP$. Acredita-se que $P \neq NP$, embora não haja prova desse fato.
Uma *transformação polinomial* é um algoritmo que dada uma instância codificada de um problema de decisão P, é capaz de transformá-la em tempo polinomial numa instância codificada P' tal que: para toda instância $s \in P$ a resposta para s é "sim", se e somente a resposta para a transformação de s em uma instância $s' \in P'$ é "sim".

Um problema de otimização combinatorial não é um problema de decisão. Mas pode ser transformado em um problema de decisão através do seguinte argumento: Seja o problema de otimização

$\min\{cx : x \in F\}$; onde x é uma representação de F.

Esse problema pode ser substituído pelo problema de decisão: "há um $x \in F$ tal que $cx \pounds k$?".
Supondo que exista um algoritmo capaz de resolver o problema de minimização em tempo polinomial, então o problema de decisão também pode se resolvido em tempo polinomial, do seguinte modo: resolve-se o problema de minimização, em seguida o de decisão, comparando a solução gerada pelo primeiro com o valor *k*.
Por outro lado, se existe um algoritmo capaz de resolver o problema de decisão em tempo polinomial, o problema de minimização também pode ser resolvido através de sucessivos questionamentos feitos para diferentes valores de *k*.
Sejam os problemas P e P'. Informalmente, uma *redução de Turing de tempo polinomial* de P em P' é um algoritmo *A* que resolve P pelo uso de uma sub-rotina hipotética *A'* para resolver e P' de tal modo que se *A'* fosse um algoritmo de tempo polinomial para P', então *A* seria um algoritmo de tempo polinomial para P.
Um problema de decisão P é dito *NP-completo,* se P pertence a *NP* e todo problema em *NP* pode ser transformado em tempo polinomial para P. Como conseqüência dessa definição, tem-se que, se um problema *NP-completo* puder ser resolvido em tempo polinomial então todos os problemas *NP* também poderão sê-lo. Neste sentido, os problemas *NP-completos,* são os problemas mais difíceis da classe *NP*.

Um problema P é chamado *NP-fácil* se existe um problema $P' \in NP$ tal que P pode ser Turing reduzido a P'. Um problema P é chamado *NP-difícil* se existe um problema de decisão P' *NP-completo* tal que P' pode ser Turing reduzido a P.
Uma visão mais aprofundada sobre complexidade pode ser vista em Garey e Johnson [Gar79] ou em Grötschel *et al* [Gro88].

2.3. O Problema do Caixeiro Viajante

Matematicamente, o PCV é descrito como um grafo $G=(V,A)$, onde $V=\{1,...,n\}$ é o conjunto dos vértices do grafo (cada um representando uma cidade) e $A=\{(i,j)\,|\,i,j=1,...,n\}$ é um conjunto de arcos ligando esses vértices (representando um caminho entre pares de cidades). Associado a cada arco existe um custo c_{ij} (distância entre as cidades), tal que $c_{ii}=\infty$. O problema consiste na determinação de um caminho *hamiltoniano* de custo mínimo sobre G.
Garey e Johnson [Gar79] demonstram que esse problema pertence à classe *NP-difícil*. Assim sendo, não existe (admitindo $P \neq NP$) um algoritmo capaz de encontrar soluções ótimas em tempo polinomial para qualquer tamanho de entrada.
Mesmo com essa perspectiva pouco otimista, é presumível que no universo dos possíveis algoritmos existam aqueles que apresentam comportamento, na maioria das situações, melhor que os demais. Este fato tem propiciado o surgimento de inúmeros procedimentos aplicáveis ao PCV.
Um dos primeiros procedimentos propostos tem origem no trabalho de Dantzig *et al* [Dan54]. Propuseram um modelo de programação inteira, associando variáveis binárias a cada arco do grafo, dado por:

Minimizar
$$\sum_{i \in V}\sum_{j \in V} c_{ij}x_{ij} \qquad (1)$$

Sujeito a
$$\sum_{j \in V} x_{ij} = 1, \quad i \in V \qquad (2)$$
$$\sum_{i \in V} x_{ij} = 1, \quad j \in V \qquad (3)$$
$$\sum_{i \in S}\sum_{j \in S} x_{ij} \leq |S| - 1, \forall S \subset V, S \neq \varnothing \qquad (4)$$
$$x_{ij} \in \{0,1\}; i,j \in V \qquad (5)$$

Nesta formulação, a parte (1) representa o custo total do circuito; as partes (2) e (3) são restrições que forçam a chegada e a saída do caixeiro viajante exatamente uma vez de cada cidade, respectivamente; a parte (4) tem como objetivo impedir a formação de sub-circuitos, figura II-1; e em (5) se $x_{ij}=1$, o trecho de i para j está no circuito, e se $x_{ij}=0$ o trecho não está;

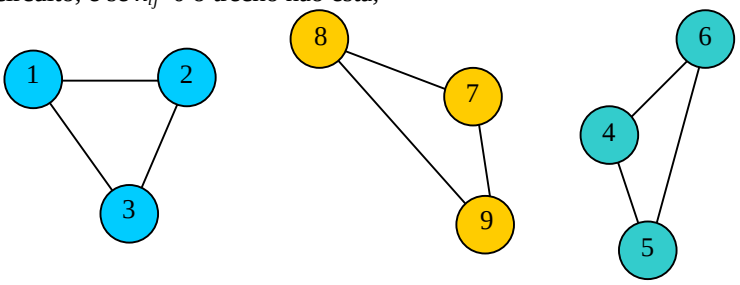


Figura III-1 : Representação de sub-circuitos

Reescrevendo essa formulação para o caso simétrico, isto é, onde $c_{ij}=c_{ji}$ para todo $i,j \in V$, tem-se

Minimizar
$$\sum_{i \in V}\sum_{j>i} c_{ij}x_{ij} \qquad (6)$$

Sujeito a
$$\sum_{j<i} x_{ij} + \sum_{j>i} x_{ij} = 2, \quad i \in V \qquad (7)$$

$$\sum_{i \in S} \sum_{\substack{j \in S \\ j>i}} x_{ij} + \sum_{i \in V-S} \sum_{\substack{j \in S \\ j>i}} x_{ij} \geq 2, \forall S \subset V, S \neq \varnothing \qquad (8)$$

$$x_{ij} \in \{0,1\}; i, j \in V, j > i \qquad (9)$$

2.4. Métodos de resolução

Os métodos desenvolvidos para resolver o PCV podem ser divididos em duas categorias:

i) *os métodos exatos* são aqueles que têm como característica a capacidade de determinar sempre uma solução ótima para o problema; e

ii) *os métodos aproximados (heurísticos)*, como o próprio rótulo já deixa claro, são aqueles que não garantem a determinação de soluções ótimas, embora eventualmente as encontrem.

2.4.1 Métodos exatos

Um procedimento clássico utilizado na resolução de diversos tipos de problemas de otimização é conhecido por *branch-and-bound.* Esse procedimento resolve problemas de otimização discreta, quebrando o conjunto de soluções viáveis em sucessivos subconjuntos menores, calculando limites inferiores para a função objetivo em cada um desses subconjuntos e, utilizando essa informação para descartar alguns desses subconjuntos de futuras considerações. Esses limites são obtidos pela substituição do problema em questão, por um conjunto de subproblemas mais fáceis de serem resolvidos (relaxações). O procedimento termina quando cada subconjunto produziu uma solução viável ou quando se demonstra que não é possível encontrar uma solução melhor que uma já em mãos. Ao final do processo, a melhor solução encontrada é uma solução ótima.

Seja um problema *P* da forma *min{g(x) / x Î T}*. Um problema *R* da forma *min{f(x)/x Î S}* é uma *relaxação* de *P* se:

i) *T Í S*; e

ii) *"x Î T* tem-se que *f(x)£g(x).*

Uma *regra de ramificação* (*branching)* é uma estratégia de particionamento do conjunto de soluções viáveis S_i de um problema P_i, em subconjuntos $S_{i1},...,S_{iq}$ tal que $\bigcup_{j=1}^{q} S_{ij} = S_i$.

Um *procedimento de determinação de limites inferiores é* um mecanismo de determinação de uma solução ótima (ou abaixo) $v(R_i)$, para a relaxação R_i de cada subproblema P_i.

Conforme descrito em Balas e Toth [Bal85], os métodos utilizando *branch-and-bound* aplicados ao PCV têm forma geral descrita por:

Passo 1 (Inicialização) Coloque o PCV em uma lista (de subproblemas ativos). Inicialize o limite superior em $U = ¥$.

Passo 2 (Seleção do subproblema) Se a lista está vazia, pare: a rota associada com *U* é ótima (se $U = ¥$, o problema não tem solução). Em outro caso, escolha um subproblema PCV_i de acordo com a regra de seleção de subproblemas e remova PCV_i da lista.

Passo 3 (Limitando inferiormente) Resolva a relaxação R_i de PCV_i ou limite $v(R_i)$ abaixo, e seja L_i o valor obtido.
Se $L_i ³ U$, retorne para o passo 2.
Se $L_i < U$ e a solução define uma rota para o PCV_i, armazene-a em lugar da melhor solução prévia, faça $U ¬ L_i$ e vá para o passo 5.

(Agora, $L_i < U$ e a solução não define uma rota)

Passo 4 (Limitando superiormente: opcional) Use uma heurística para encontrar uma rota para o PCV. Se uma rota melhor que a corrente é encontrada, armazene-a em lugar da última e atualize *U*.

Passo 5 (Redução: opcional) Remova do grafo de PCV_i todos os arcos cuja inclusão em uma rota elevariam seu valor acima de *U*.

Passo 6 (Ramificando) Aplique a regra de ramificação ao PCV_i, isto é, gere um novo conjunto de subproblemas PCV_{i1}, ..., PCV_{iq}, coloque-os na lista, e vá para o passo 2.

- *Relaxação para o Problema de Designação.*

A mais direta relaxação para a formulação proposta por Dantzig *et al* [Dan54], consiste na remoção das restrições que impedem a formação de sub-circuitos (equação (4)). Assim procedendo, o problema resultante é transformado em um *Problema de Designação* (*Assignament Problem*) - *(PD).* Esse problema consiste na determinação de um emparelhamento (*matching*) de dois subconjuntos de vértices de mesma cardinalidade, de tal modo que a soma dos pesos associados a cada um dos arcos utilizados no emparelhamento tenha custo mínimo. O *PD* pode ser resolvido eficientemente através do *Método Húngaro*, vide Christofides [Chr75].

Diferentes regras de ramificação podem ser utilizadas com um tipo de relaxação. Uma boa ramificação deve possuir duas características:

i) gerar poucos sucessores para cada nó da árvore de busca; e

ii) gerar subproblemas fortemente restritos, isto é, deve excluir muitas soluções de cada subproblema.

Uma regra simples de ramificação para um subproblema consiste na formulação de dois novos subproblemas, utilizando as mesmas restrições do problema a ser ramificado, acrescido de novas restrições. Um dos novos subproblemas deve conter uma restrição que obrigue a presença de um determinado arco na solução, enquanto no outro deve existir uma restrição que impeça a presença desse mesmo arco.

De modo geral, sejam E_k e I_k os conjuntos de arcos excluídos e incluídos na solução de um subproblema *k*, respectivamente. Então, um subproblema *k* pode ser definido pelo conjunto de equações de (1) a (5) mais as restrições

$$x_{ij} = \begin{cases} 0, & (i,j) \in E_k \\ 1, & (i,j) \in I_k \end{cases}. \qquad (10)$$

Uma relaxação desse subproblema é formada pelas equações (1), (2), (3), (5) e (10). Várias dessas regras podem ser encontras em Balas e Toth [Bal85].

- *Relaxação para uma 1-Tree com Função Objetivo Lagrangeana.*

Seja um PCV representado pelo grafo *G(V, A).* Seja um subgrafo *H* de *G* conectado e com *n* arcos, formado pela expansão de uma árvore sobre *G* mais um arco. A classe formada pelos subgrafos *H*, que contém um vértice de grau dois e está contido em um único ciclo de *H* é denominada *classe 1-tree*.

Seja o vértice de grau dois de *H* o vértice 1. Uma relaxação *1-tree* é obtida com

$$\sum_{i \in S} \sum_{\substack{j \in V'-S \\ j>i}} x_{ij} + \sum_{i \in V'-S} \sum_{\substack{j \in S \\ j>i}} x_{ij} \geq 1, \forall S \subset V' = V - \{1\}, S \neq \emptyset \qquad (11)$$

$$\sum_{i \in V} \sum_{j>i} x_{ij} = n \qquad (12)$$

$$\sum_{j \in V} x_{1j} = 2 \qquad (13)$$

mais as equações (6) e (9).

O problema de encontrar uma *1-tree* de mínimo custo pode ser decomposto em dois problemas independentes:

i) encontrar uma árvore de custo mínimo sobre *G-{1}*; e

ii) encontrar os dois arcos de menor custo incidentes ao vértice *1* em *G*.

Essa relaxação pode ser fortemente restringida pela introdução da equação (7) na função objetivo gerando uma *função lagrangeana* e então, maximizando-a em função dos seus multiplicadores.

Tal relaxação fica definida pela a equação:

$$\begin{aligned} L(l) &= \min_x \left\{ \sum_{i \in V} \sum_{j>i} c_{ij} x_{ij} + \sum_{i \in V} l_i \left(\sum_{j<i} x_{ji} + \sum_{j>i} x_{ij} - 2 \right) \right\} \\ &= \min_x \left\{ \sum_{i \in V} \sum_{j>i} (c_{ij} + l_i + l_j) x_{ij} \right\} - 2 \sum_{i \in V} l_i \end{aligned} \qquad (14)$$

sujeita às restrições (9), (11), (12) e (13), onde l_k é um vetor qualquer. Essa formulação é denominada *relaxação lagrangeana do PCV*.

Para qualquer l, tem-se que $L(l) \leq v(PCV)$. Então a mais forte relaxação é dada por $\bar{l}$, tal que

$$L(\bar{l}) = \max_{l}\{L(l)\}. \qquad (15)$$

Essa equação é conhecida como função *Lagrangeana dual.*
Pode-se utilizar para resolver a função *Lagrangeana dual* o método de otimização por subgradiente, utilizado pela primeira vez para resolver o PCV por Held e Karp [Kar71].
O algoritmo de otimização por subgradiente é descrito a seguir:

Passo 1 Inicie com algum $l = l^0$, $k=1$;

Passo 2 Resolva $L(l^k)$.

Passo 3 Se $H(l^k)$ a 1-tree ótima encontrada é uma rota valida ou se $v(H(l^k)) \geq U$ então Pare.

Passo 4 Faça $l_i^{k+1} = l_i^k + t^k(d_i^k - 2), i \in V$, *onde* t^k é o comprimento do passo definido por $t^k = \alpha(U - L(l^k))/\sum_{i \in V}(d_i^k - 2)^2$, onde $0<\alpha\leq 2$ e d_i é grau do vértice i em $H(l^k)$.

Passo 5 Faça $k=k+1$. Vá para o passo 2.

Pode ser demonstrado que a seqüência $L(l^k)$ converge para o valor ótimo $L(l^*)$ se $\sum_{k=1}^{\infty} t^k \rightarrow \infty$ e $\lim_{k\rightarrow\infty} t^k = 0$.

O estado da arte entre os algoritmos exatos é formado por algoritmos que utilizam a abordagem de *branch-and-cut*. Um algoritmo de *branch-and-cut* é um algoritmo de *branch-and-bound* no qual *planos cortantes* são gerados ao longo da árvore de busca. A mudança provocada por essa filosofia está no fato de que a busca por soluções rápidas em cada nó é substituída pela procura por limites mais apertados. Com esse tipo de procedimento, problemas com mais de duas mil cidades passaram a ser resolvidos para o ótimo.

Padberg e Rinald [Pad91] resolveram o problema PR2392 da TSPLIB usando um procedimento com esse tipo de abordagem. Para isso, utilizaram hardware poderoso e software extremamente complexo. Falando a respeito da complexidade do código, citam que “o programa é composto por cerca de 120 rotinas com aproximadamente 8.500 linhas, não incluindo os comentários e o programa responsável pela resolução do problema de programação linear”. O tempo gasto pelo algoritmo para a resolução do problema foi de 4,3 horas (“rodando” em um IBM 3090/600). A relação entre tempo de execução do procedimento e tamanho do problema apresentou comportamento exponencial. Portanto, limitando a utilização do procedimento a supercomputadores, no caso de problemas com muitas cidades.

Mais recentemente, Applegate *et al* [App95] determinaram o valor ótimo para diversos problemas da TSPLIB com tamanhos variando entre 225 a 7397 cidades. Para isso utilizaram uma rede de estações de trabalho UNIX (não dão detalhes sobre a quantidade ou tipo das estações e nem sobre o tempo gasto), utilizando um procedimento que é uma combinação de diversos algoritmos, inclusive o de Padberg e Rinald [Pad91]. Johnson e McGeoch [Joh97] dão uma pista sobre o tempo utilizado: "3 a 4 anos do tempo de CPU de uma rede de máquinas do porte de uma SPARCStation 2".
O avanço histórico observado na resolução de PCV's simétricos com matrizes de distâncias não aleatórias para o ótimo e em função do número de cidades, foi: 49 em 1954 [Dan54]; 120 em 1977 [Gro80]; 318 em 1980 [Cro80]; 2392 em 1988 [Pad91]; e 7397 em 1995 [App95].
Entretanto, existem problemas práticos que apresentam grandeza na ordem de dezenas de milhares de vértices, ou seja, muito acima desses limites. Junger *et al* [Jun93], falando das dificuldades de utilização de métodos exatos, declaram que "pesquisadores interessados em resolver problemas práticos com esse tipo de procedimento, têm descrito tais algoritmos como impraticáveis".

2.4.2 Métodos heurísticos

Desde que os métodos exatos são do ponto de vista computacional muito dispendiosos, alternativas mais econômicas têm sido procuradas. Deste esforço, uma grande variedade de *heurísticas* com alta eficiência têm surgido.
"Uma *heurística* é uma técnica que busca boas soluções, isto é, soluções próximas do ótimo, com um custo computacional razoável sem garantir a otimalidade, e possivelmente a viabilidade. Inoportunamente pode não ser possível determinar quão próximo uma *solução heurística* em particular está da solução ótima", Reeves [Ree96].
Na prática, a despeito dessa conceituação pessimista, técnicas heurísticas têm sido utilizadas com bastante sucesso em vários tipos de problemas.
Mais adiante serão apresentadas algumas dessas técnicas. Antes disso, uma breve introdução sobre conceitos básicos de busca local será feita, pois é nesse universo que todas essas técnicas estão imersas.

2.4.2.1 Busca local

Busca local, também referida na literatura como busca na vizinhança, é a estratégia base de muitos dos métodos heurísticos utilizados na solução de problemas de otimização.
Seja um problema de otimização representado pelo par (*S*,*g*), onde *S* é um conjunto de soluções viáveis[2] e *g* a função objetivo que associa cada elemento $s \in S$ a um número real. Em um problema de minimização o objetivo é encontrar um elemento $s^* \in S$ tal que $g(s^*) \leq g(s) \quad \forall s \in S$.

Uma *vizinhança* é definida através de uma função $N : S \rightarrow 2^S$, que associa cada elemento $s \in S$ a um conjunto de soluções alcançáveis com um movimento simples. Entenda-se como movimento simples, uma pequena modificação aplicada em uma solução corrente.
No caso do PCV, por exemplo, para uma solução corrente viável (uma rota válida) a vizinhança pode ser definida como: "todas as rotas geráveis pela substituição de dois arcos presentes na rota corrente por outros dois arcos que não estão nesta rota e que gerem também uma rota viável".
Uma solução *x* é chamada de mínimo local com respeito a *g*, se $g(x) \leq g(y), \forall y \in N(x)$.
Algoritmos de busca local são processos de otimização iterativos. Iniciam com uma solução, e iterativamente, procuram na vizinhança desta solução uma outra de menor custo (ou que gere expectativa de ganho futuro). Se tal solução é encontrada a solução inicial é substituída por esta, e a busca continua.
Alguns dos métodos baseados em busca local são: têmpera simulada, algoritmos genéticos, busca tabu, busca local dirigida e busca gulosa entre outros. Mais adiante alguns desses métodos serão apresentados, bem como sua forma de aplicação ao PCV.

2.4.2.2 Métodos de construção

Os métodos mais elementares para geração de soluções para o PCV são denominados métodos de construção. Esses métodos procuram encontrar uma solução próxima do ótimo partindo simplesmente da matriz de distâncias. Entre outros se destacam [Bod83]:
(i) *Procedimento do vizinho mais próximo* [Ros77].

[2] Denomina-se solução viável, a toda solução que satisfaz todas as restrições do problema em questão.

Passo 1 Inicie o caminho com um vértice qualquer do grafo.

Passo 2 Encontre o vértice mais próximo do último vértice incluído no caminho e inclua-o no caminho.

Passo 3 Repita o passo 2, até que todos os vértices tenham sido incluídos.

Passo 4 Conecte o último vértice incluído no caminho ao vértice inicial.

(ii) *Economia de Clark e Wright* [Cla64].

Passo 1 Selecione qualquer vértice como nó inicial.

Passo 2 Compute as economias $s_{ij} = c_{1i} + c_{1j} - c_{ij}; i, j = 2,...,n$

Passo 3 Ordene as economias da maior para a menor.

Passo 4 Iniciando no topo da lista de economias, mova-se para baixo formando circuitos maiores pela junção apropriada de vértices *i, j*.

Passo 5 Repita o passo 4, até que todo o circuito esteja formado.

(iii) *Procedimentos de inserção* [Ros77].

Neste grupo de procedimentos, quando da inserção de um novo vértice, primeiro procura-se determinar qual o vértice que será adicionado ao caminho, e em seguida qual o local mais apropriado para fazer essa inserção.

iii.a) *Inserção mais barata*

Passo 1 Inicie o caminho com um vértice qualquer do grafo. Seja esse vértice o vértice *i*.

Passo 2 Encontre o vértice *k*, mais próximo de *i*, e forme um sub-circuito *i-k-i*.

Passo 3 Encontre o arco (*i,j*) no sub-circuito e um vértice *k* fora dele, tal que $c_{ik} + c_{kj} - c_{ij}$ seja mínimo. Insira *k* entre *i* e *j*, formando o sub-circuito... *-i-k-j-...* .

Passo 4 Pare se todos os vértices já estão inseridos. Caso contrário volte para o passo 3.

iii.b) *Inserção do mais próximo*

Passo 1 Inicie o caminho com um vértice *i* qualquer do grafo.

Passo 2 Encontre um vértice *j* mais próximo de *k* (c_{ik} é mínimo) e forme um sub-circuito *i-k-i*.

Passo 3 Em relação ao sub-circuito corrente, encontre um vértice *k* que ainda não esteja nele e seja mais o próximo[3].

Passo 4 Encontre o arco (*i,j*) no sub-circuito, tal que minimize $c_{ik} + c_{kj} - c_{ij}$. Insira então *k* entre *i* e *j* formando o sub-circuito *...-i-k-j-...* .

[3] Seja C_p o conjunto dos vértices que já estão no circuito, e C_p' os que não estão no passo *p*. Define-se a distância de um vértice $k \in C_p$' para um sub-circuito C_p por $d_k^p = \min\{d_{ik} \mid i \in C_p\}$. O vértice mais próximo é aquele cujo d_k^p é mínimo.

Passo 5 Pare se todos os vértices já estão inseridos. Caso contrário volte para o passo 3.

iii.c) *Inserção arbitrária*

O mesmo que na inserção do mais próximo sendo que o passo 3 é substituído por:

Passo 3' Selecione um vértice *k* qualquer que não esteja no sub-circuito e o adicione ao sub-circuito.

iii.d) *Inserção do mais distante*

Idêntico ao procedimento (iii.b), substituindo a expressão "mais próximo" nos passos 2 e 3 pela expressão "mais distante".

iii.e) *Envoltória convexa*

Passo 1 Construa uma envoltória convexa sobre o conjunto de vértices. A envoltória formada determina o sub-circuito inicial.

Passo 2 Para cada vértice *k* que não está no sub-circuito, encontre um par (*i*,*j*) tal que $c_{ik} + c_{kj} - c_{ij}$ é mínimo.

Passo 3 De todos as ternas (*i*,*k*,*j*) encontradas no passo 2, determine (i^*,k^*,j^*) de modo que $(c_{i^*k^*} + c_{k^*j^*}) / c_{i^*j^*}$ é mínimo.

Passo 4 Insira o vértice k^* entre os vértices i^* e j^*.

Passo 5 Repita os passos de 2 a 4 até obter um circuito com todos os vértices.

Golden *et al* [Gol80] realizaram estudo comparativo dessas heurísticas aplicadas a instâncias com 100 vértices. Nesse estudo, as heurísticas *inserção do vizinho mais próximo*, *inserção do mais próximo* e *inserção mais econômica* geraram soluções com excesso[4] médio, com relação às melhores soluções conhecidas[5], maior que 11%. As heurísticas de *Clark-Wright* (*CW*), *inserção do mais distante* e *inserção arbitrária* geraram soluções mais satisfatórias, com excesso médio em torno de 4%. A *envoltória convexa*, dentre todas, foi a que apresentou os melhores resultados, com excesso médio na casa de 3%. As heurísticas mais eficientes se comportaram de modo similar aos procedimentos *2-opt* (melhor resultado em 25 rodadas) e *3-opt* (uma rodada) quando iniciando de soluções aleatórias. Tais procedimentos serão descritos mais adiante no tópico 2.4.2.3.1.

2.4.2.3 Métodos de melhoramento

Neste tipo de procedimento, uma rota inicial viável é obtida através de algum mecanismo. Usualmente, utiliza-se um dos métodos descritos anteriormente ou uma rota aleatória qualquer. O procedimento de melhoramento é então aplicado sobre essa rota inicial gerando como conseqüência uma nova rota mais próxima da ótima.

2.4.2.3.1 Procedimentos k-Opt

[4] Definição na seção 4.5.

Um conjunto importante de heurísticas desenvolvidas para o PCV é formado pelas trocas *k-opt*. Em linhas gerais tratam-se de algoritmos que partindo de uma solução inicial viável para o PCV, realizam permutas de *k* arcos que estão no circuito por outros *k* arcos fora deste, buscando a cada troca diminuir o custo total do circuito.
O primeiro mecanismo de trocas, denominado troca *2-opt*, foi proposto por Croes [Cro58]. Neste mecanismo, uma nova solução é gerada através da remoção de dois arcos, resultando em dois caminhos. Um dos caminhos é invertido e em seguida reconectado para formar uma nova rota, conforme mostra a figura II-2. Essa nova solução passa a ser a solução corrente e o processo se repete até que não seja mais possível realizar uma troca de dois arcos com ganho. Daí o nome troca *2-opt*, *opt* de *optimum*, indicando que ao término do processo a solução resultante não pode mais ser melhorada com qualquer troca de dois arcos.
Em 1965, Shen Lin [Lin65] propõe uma ampliação da vizinhança de busca. No caso das trocas *2-opt*, uma solução corrente tem sua vizinhança representada por todas as soluções alcançáveis, considerando a troca

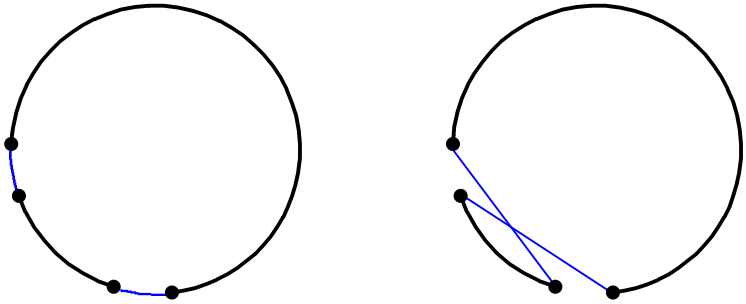


Figura III-2 : Movimento *2-opt.*

de dois arcos. Lin propõe uma vizinhança mais alargada ao considerar todas as soluções alcançáveis com uma troca de três arcos, conforme mostra a figura II-3. Denominou esse mecanismo de troca *3-opt*. Em sua avaliação da qualidade desse procedimento, observou que, em média, os resultados gerados por trocas *3-opt* eram consideravelmente melhores que os da troca *2-opt*, e que a probabilidade de se encontrar valores ótimos também é muito maior. Observou também, que o tempo de execução da busca *3-opt* era maior que o da *2-opt* por um fator de 5.
Sugeriu ainda, que durante o processo de busca das trocas, dever-se-ia considerar a primeira troca favorável em qualquer estágio, em vez de buscar a troca de maior ganho possível na vizinhança (busca

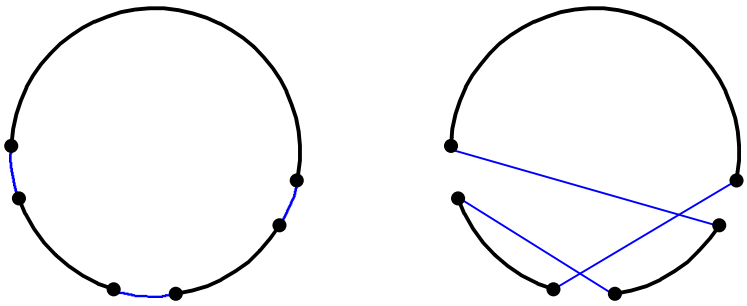


Figura III-3 : Movimento *3-opt.*

gulosa), pois o tempo necessário para esse tipo de busca seria demasiadamente alto.
Em 1973, Lin e Kernighan [Lin73] propõem um algoritmo onde o número de arcos trocados em cada passo é variável. As trocas de arcos são realizadas segundo um critério de ganho que restringe o tamanho da vizinhança de busca. As soluções produzidas são de alta qualidade, superando as obtidas através de trocas *3-opt* padrão. Pela qualidade dos resultados gerados, durante muitos anos, diversos autores o mencionaram como o campeão absoluto entre os métodos heurísticos. Passo a passo, o algoritmo é:
Passo 1 Gere um circuito aleatório T inicial.
Passo 2 Faça $G^*=0$. [G^* representa o melhor ganho alcançado até o momento]. Escolha qualquer nó t_1 e seja x_1 um dos arcos de T adjacentes a t_1. Faça $i=1$.

---

[5] Soluções encontradas usando procedimento *k-opt* e apresentadas por Lin e Kernighan [Lin73].

Passo 3 Do outro ponto final t_2 de x_1 escolha y_1 para t_3 com $g_1=|x_1|-|y_1|>0$. Se não existe y_1, vá para o passo 6(d).

Passo 4 Faça $i=i+1$. Escolha x_i [que no momento liga t_{2i-1} a t_{2i}] e y_i como segue:

(a) x_i é escolhido de modo que, se t_{2i} é ligado com t_1, a configuração resultante é um circuito.

(b) y_i é algum arco disponível no ponto final t_{2i} compartilhado com x_i, sujeito a (c), (d) e (e). Se não existir y_i, vá para o passo 5.

(c) Para garantir que os x's e y's são disjuntos, x_i não pode ser um arco previamente unido (isto é, um y_j, $j<i$), e similarmente y_i não pode ser um arco previamente quebrado.

(d) $G_i = \sum_{j=1}^{i} g_j = \sum_{j=1}^{i} (|x_j| - |y_j|) > 0$. [Critério de ganho].

(e) Em seqüência para garantir que o critério de viabilidade de (a) possa ser satisfeito para $i+1$, o y_i escolhido deve permitir a quebra de x_{i+1}.

(f) Antes de y_i ser construído, deve-se verificar se fechando a ligação de t_{2i} par t_1 será observado ganho maior que o melhor alcançado anteriormente. Faça y_i^* ser um arco conectando t_{2i} a t_1 e faça $g_i^*=|y_i^*|-|x_i|$. Se $G_{i-1}+g_i^*>G^*$, faça $G^*=G_{i-1}+g_i^*$ e $k=i$.

Passo 5 Terminada a construção de x_i e y_i dentro dos passos de 2 até o 4 quando não houver mais arcos satisfazendo a os critérios de 4(c) a 4(e), ou quando $G_i £ G^*$. Se $G^*>0$ então considere o novo circuito T' faça $f(T')=f(T)-G^*$ e $T \neg T'$ e vá para o passo 2.

Passo 6 Se $G^*=0$, um recuo limitado é realizado como segue:

(a) Repita os passos 4 e 5 escolhendo y_2's na ordem crescente do comprimento. Contanto que o critério de ganho $g_1+g_2>0$ seja satisfeito.

(b) Se todas as escolhas de y_2 no passo 4(b) são exauridas sem ganho, retorne ao passo 4(a) e tente escolher outro x_2.

(c) Se isto também falha, um passo atrás no passo 3 é promovido, aonde os y_i's são examinados na ordem do aumento do comprimento.

(d) Se os y_i's são também exauridos sem ganho, tenta-se alternar x_1 no passo 2.

(e) Se isto também falha, um novo t_1 é selecionado, e repete-se o passo 2.

Passo 7 O procedimento termina quando todos os n valores de t_1 tiverem sido examinados sem ganho.

Em linhas gerais, o algoritmo *LK* parte de um circuito inicial T e de um vértice inicial t_1. Estando no passo i, uma troca é feita através da seqüência: remove-se o arco (t_1, t_{2i}), e adiciona-se o arco (t_{2i}, t_{2i+1}); o arco (t_{2i+1}, t_{2i+2}) é escolhido para ser removido, se com sua remoção e com a adição do arco (t_{2i+2}, t_1) um circuito seja formado. O arco (t_{2i+2}, t_1) é removido se e quando o passo $i+1$ é executado. Uma representação esquemática para o movimento pode ser vista para os níveis 1, 2, e 3, na figura II-4.

O número de inserções e remoções no processo de busca é limitado pelo critério de ganho de *LK*. Esse critério, basicamente limita a seqüência de trocas àquelas que gerem ganhos positivos (redução do tamanho da rota) a cada passo da seqüência, isto é, são desconsideradas, em princípio, seqüências onde algumas trocas resultem em ganhos positivos e outras em ganhos negativos, mas cujo ganho total da seqüência seja positivo.

Um outro mecanismo presente no algoritmo é a analise de soluções alternativas nos níveis 1 e 2, sempre que uma nova solução é gerada com ganho zero (passo 6). Isso é feito através da escolha de novos arcos

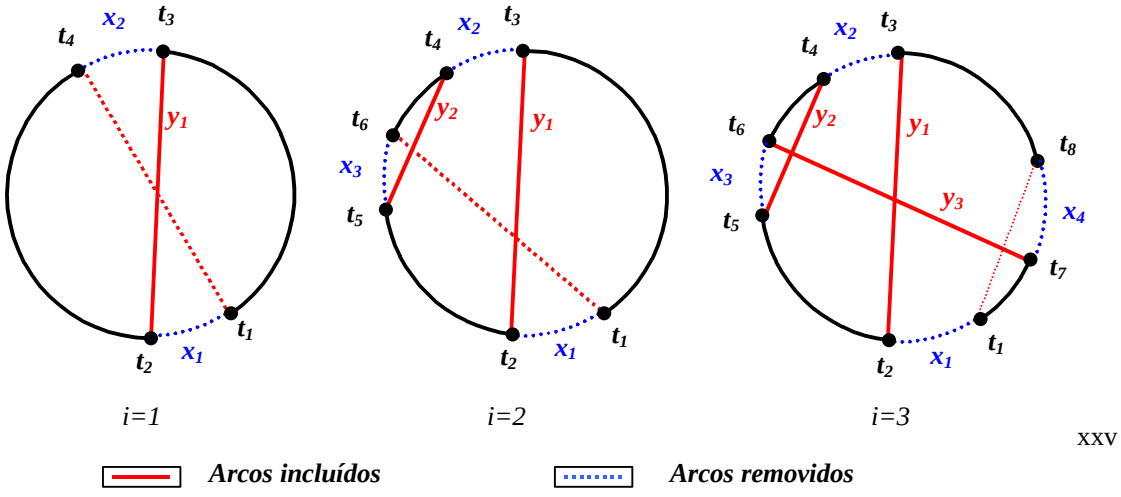


Figura III-4: Movimento do algoritmo *LK* para os níveis 1,2, e 3.

para troca.
Duas variantes do algoritmo *LK* são os algoritmos *LK-repetido* e o *LK-iterado (LKI).* No primeiro, *r* diferentes circuitos iniciais são considerados e o *LK* é aplicado a cada um deles. O algoritmo produz como resultado a melhor solução encontrada entre todas as repetições. O segundo, como descrito em Johnson e McGeoch [Joh97], tem a forma:

Passo 1 Gere um circuito aleatório inicial T.

Passo 2 Para um número M especificado repita:
2.1) Realize um movimento aleatório não viciado 4-opt (troca de quatro arcos) sobre T, obtendo T'.
2.2) Execute o *LK* sobre T', obtendo T''.
2.3) Se comprimento(T'')≤comprimento(T'), faça T=T''.
Passo 3 Retorne T.

Hoje o *LKI* é considerado um dos melhores algoritmos heurísticos para o PCV.
Os procedimentos de trocas de arcos podem ser acelerados de modo substancial para problemas PCV com a utilização de listas de vizinhos para cada vértice do grafo. Isto é feito construindo-se uma lista com os *k* vértices mais próximos de um vértice t_2, dispostos em ordem crescente de distâncias para t_2. Os candidatos na troca de um arco (t_1, t_2) serão aqueles arcos (t_2, x), tais que $|(t_2, x)|<|(t_2, t_1)|$ e x está na lista de t_2. A eleição dos candidatos é feita através da varredura dessa lista do primeiro até o elemento y, que satisfaça a expressão $|(t_2, y)| \geq |(t_2, t_1)|$. Os elementos que antecederem a y são os candidatos à troca.
Além de listas de vizinhos, estruturas de dados especiais têm sido desenvolvidas com o objetivo de acelerar a busca, usando procedimentos *r-opt*. Em Fredman et al [Fre95] são testadas as estruturas *Splay Tree*, *Two-level Tree*, *Segment Tree* para a representação de rotas. Os autores argumentam que um tempo considerável é desperdiçado no processo de inversões de segmentos presentes em algoritmos de troca de arcos, quando utilizada a representação de rotas através de vetores. O efeito torna-se mais significativo à medida que cresce o número de cidades envolvidas no problema. Os resultados apresentados, comparando o desempenho das diferentes estruturas, indicam uma redução no tempo de execução do algoritmo *LK* bastante significativa em relação a uma versão, na qual se utiliza a representação por vetores. Em problemas com até 1.000 cidades as diferenças são pequenas, a ponto de algumas das estruturas chegarem a apresentar resultados ligeiramente piores que a representação por vetores. Contudo, com o aumento do número de cidades, passam a apresentar uma redução no tempo de pelo menos 30% para instâncias com 10.000 cidades e de 90% para problemas com 100.000 cidades, se comparadas à representação por vetores. Estimam também uma redução no pior caso (usando *segment tree*) em 97% e de 98,3% no melhor caso (usando *two-level tree*) para problemas com 1.000.000 de cidades.
Ainda com relação à aceleração do processamento, existe uma técnica proposta por Bentley [Ben92] apud [Vou97] conhecida como "*Don't look bits*". Baseia-se na seguinte idéia: se um vértice t_1, em uma fase de busca prévia do processo, não permitiu um ganho e nem os seus vizinhos no circuito também foram modificados, é porque, provavelmente, não será encontrado movimento de melhora envolvendo-o, portanto no passo presente ele não deve ser olhado. Para executar esse procedimento, utiliza-se uma marcação (um bit) para cada um dos vértices. Inicialmente, todas as marcas estão no estado *desligado*, permitindo que todos os vértices sejam explorados. Altera-se a marcação de um vértice t_1 para *ligado*, sempre que um processo de troca iniciando em t_1 é tentado, e essa tentativa falha. A alteração para *desligado é* feita sempre para os pontos finais dos arcos removidos em uma troca.
Um comparativo do desempenho dos métodos de *CW, inserção do mais próximo, 2-opt e 3-opt* implementados com listas de vizinhos, pode ser visto na tabela II-2. Do mesmo modo, um comparativo dos procedimentos *LK-repetido* e *LK-iterado* na tabela II-3.

Excesso médio das soluções geradas pelas heurísticas *2-opt, 3-opt, CW* e inserção do mais próximo sobre o limite inferior de Held-Karp.

n	*100*	*1.000*	*10.000*	*100.000*
Instâncias com distâncias Euclidianas				
Inserção do mais próximo	19,5	17,0	16,6	14,9
CW	9,2	11,3	11,9	12,1
2-opt	4,5	4,9	5,0	4,9
3-opt	2,5	3,1	3,0	3,0
Matriz de distâncias aleatórias				

Excesso médio das soluções geradas pelas heurísticas *2-opt, 3-opt, CW* e inserção do mais próximo sobre o limite inferior de Held-Karp.

n	*100*	*1.000*	*10.000*	*100.000*
Instâncias com distâncias Euclidianas				
Inserção do mais próximo	19,5	17,0	16,6	14,9
CW	9,2	11,3	11,9	12,1
2-opt	4,5	4,9	5,0	4,9
3-opt	2,5	3,1	3,0	3,0
Inserção do mais próximo	100	170	250	-
2-opt	34	70	125	-
3-opt	10	33	63	-

* *Adaptada de Johnson e McGeoch [Joh97*]

Tabela III-2

Na tabela II-2, observa-se que os ganhos gerados pela utilização de procedimentos de *2-opt* e *3-opt* são significativos se comparados com a utilização de procedimentos de construção simples como inserção do mais próximo e *CW*. A distância média para o limite inferior de Held-Karp passa a ser 5% no caso do *2-opt*, e 3% para *3-opt*. Enquanto isso, a aproximação média é de 11% para CW e 17% para inserção do mais próximo.

Tabela III-3
Desempenho do *LK-iterado* frente ao *LK-repetido*: Excesso médio sobre os limites de Held-Karp e tempos médios de execução, dado o tamanho do problema.

	Excesso médio (%)				*Tempo médio (segundos)*			
n	*100*	*1.000*	*10.000*	*100.000*	*100*	*1.000*	*10.000*	*100.000*
LK-repetido: gerações independentes.								
1	1,52	2,01	1,96	1,95	0,06	0,8	10	150
n/10	0,99	1,41	1,71	-	0,42	48,1	7.250	-
$n/10^{.5}$	0,92	1,35	1,68	-	1,31	151,3	22.900	-
n	0,91	1,29	1,65	-	4,07	478,1	72.400	-
LK-iterado: iterações.								
n/10	1,06	1,25	1,26	1,31	0,14	5,1	189	10.200
$n/10^{.5}$	0,96	0,99	1,04	1,08	0,34	13,6	524	30.700
n	0,92	0,91	0,89	-	0,96	39,7	1.570	-

* *Adaptada de Johnson e McGeoch [Joh97]*

Comparando-se os resultados obtidos pelos algoritmos *LK*-iterado e *LK*-repetido, observa-se uma clara superioridade do primeiro. Quando o número de cidades aumenta, além de gerar soluções melhores, o tempo necessário para isso também é menor.

Se comparados os resultados das tabelas II-2 e II-3, o ganho produzido pela aplicação dos algoritmos *LK* fica evidente.

2.4.2.3.2 Algoritmos genéticos

Na década de 70, uma nova categoria de algoritmos evolucionários, que ficou conhecida como *Algoritmos Genéticos* (*AG*), foi proposta por Holland [Hol75]. Em relação aos estudos mais primitivos, ele acrescentou uma nova maneira de gerar indivíduos, através de um mecanismo de reprodução

conhecido como *reprodução sexuada*. Nesse tipo de reprodução, novos indivíduos eram gerados através do cruzamento (*crossover*) de dois outros indivíduos da população corrente, isto é, novos indivíduos eram formados a partir da combinação do "material genético" de cromossomos pais.

2.4.2.3.2.1 Princípios básicos

Os *AG* procuram solucionar um dado problema, partindo de uma população inicial de soluções potenciais. Esse multiconjunto de soluções (população) é sistematicamente modificado através de determinadas regras básicas, que procuram "imitar" os princípios gerais que regem a evolução natural das espécies. Basicamente, esses princípios falam da sobrevivência do mais adaptado, isto é, da forma como o meio ambiente seleciona os seres mais aptos. Em geral, estes conseguem se reproduzir mais facilmente que os menos adaptados. Portanto, as diferenças que facilitam a sobrevivência desses seres são transmitidas com maior freqüência através da herança genética às futuras gerações. Ao longo das gerações, essas características firmam-se propiciando um melhoramento global da população. Em outras palavras, características genéticas presentes em indivíduos com alto grau de adaptação são mais intensamente transmitidas para as gerações seguintes. Ao longo do tempo, essas características favoráveis tornam-se cada vez mais presentes nos indivíduos, enquanto as desfavoráveis vão sendo eliminadas. Como resultado, esse processo conduz a um melhoramento global do nível de adaptação dos indivíduos presentes em populações futuras.

Esses princípios são utilizados em algoritmos genéticos da seguinte maneira: determinando-se o espaço de soluções *S* para o problema de otimização que se deseja resolver; construindo-se uma população inicial de soluções; e aplicando-se um conjunto de operadores que simulam os fenômenos de transmissão de genes e de geração de novos indivíduos. A forma de representação dos elementos de *S* e os mecanismos de transmissão de genes podem ser implementados de diferentes modos.

Em geral, AG's representam os elementos de *S* através de palavras (strings) de comprimento *n*, sobre um alfabeto *A={0, 1,..., k-1}*, sendo cada palavra denominada *cromossomo*. Cada posição é chamada *loci*. A variável em cada posição é chamada *gene*, e seu valor individual um *allele*. O conjunto de cromossomos é chamado *genótipo*, o qual define um *fenótipo* (o indivíduo) com um certo ajuste (*fitness*).

De modo ideal, deseja-se que haja uma correspondência biunívoca entre cada solução do problema e cada elemento de *S*, o que nem sempre é possível.

Inicialmente, um multiconjunto de cromossomos é selecionado de *S*, formando uma *população*. Inicia-se então a busca por boas soluções na população, através da avaliação do ajustamento de cada um de seus elementos. São selecionados os cromossomos pais, aplicados operadores de *crossover* sobre eles e de mutação nos filhos gerados. Como resultado dessas operações, uma nova população é formada. De modo repetido, segue-se nesse processo até que um critério de parada seja alcançado.

Passo a passo, um algoritmo genético pode ser descrito como:

Passo 1	Crie uma população inicial de P cromossomos (geração 0).
Passo 2	Avalie o ajuste de cada cromossomo.
Passo 3	Selecione os pais na população corrente através de um mecanismo de seleção que privilegie a reprodução dos cromossomos mais ajustados.
Passo 4	Escolha um par de pais aleatoriamente para cruzamento. Cruze os cromossomos em um ponto de corte de modo a formar dois novos descendentes para a população.
Passo 5	Execute mutações nos descendentes e insira o indivíduo resultante dessa mutação na nova população.
Passo 6	Repita os passos 4 e 5, até que todos os pais sejam selecionados e combinados (P descendentes serão gerados).
Passo 7	Substitua a velha população de cromossomos pela nova geração.
Passo 8	Avalie o ajuste de cada cromossomo na nova população.
Passo 9	Volte para o passo 3 até que um limite superior de gerações tenha sido alcançado. Caso contrário, exiba o melhor cromossomo obtido na busca.

Antes de abordar mais detalhadamente os elementos introduzidos no algoritmo, seja o seguinte problema ilustrativo:

Minimizar $f(x) = |x - 20|$; sujeito a $x \in \{0,1,...63\}$.

Cada valor de *x*, isto é cada solução pode ser representada conforme tabela II-4:

x	*Cromossomo*
0	000000
1	000001
2	000010
...	...
62	011111
63	111111

Tabela III-4 : Representação de soluções

Neste caso, o conjunto de soluções *S* é representado por palavras de comprimento *n=6*, sobre o alfabeto *A={0,1}*. Suponha que tenham sido aleatoriamente escolhidas para formar a população inicial as soluções dispostas na tabela II-5.

x	*Cromossomo*	*Ajustamento (f(x))*
1	000001	19
6	000110	14
15	001111	5
24	011000	4
26	011010	6
34	100010	14
53	110101	33

Tabela III-5 : Avaliação de soluções

Verifica-se nesta tabela, que as soluções com melhor ajuste são 15, 24 e 26 com ajustamento igual a 5, 4 e 6 respectivamente, e que os cromossomos que representam essas soluções tem em comum a terceira posição da esquerda para direita ocupada pelo valor 1. Assim, alguém poderia questionar se a solução ótima teria ou não essa mesma característica.
AG's exploram essas similaridades. Desta forma, em uma próxima geração se privilegiará a formação de soluções que apresentem esse tipo de característica comum aos indivíduos mais ajustados da população corrente e, provavelmente, também a solução ótima.

2.4.2.3.2.2 Evolução da população

Tradicionalmente, dois tipos de processos de substituição da população têm sido utilizados. O primeiro é chamado de *Evolução Geracional.* E é caracterizado pela substituição da população corrente a cada geração. Uma opção freqüentemente usada para minimizar o rompimento com a população corrente, é a de transferir para a próxima geração os melhores indivíduos da população corrente. O segundo é denominado *Evolução de Estado Firme*. Nesse, os novos cromossomos gerados são inseridos na população corrente em substituição aos cromossomos de pior ajuste. A vantagem desse procedimento, é que novos cromossomos de alto potencial passam a contribuir imediatamente para a qualidade da população. A performance entre os dois procedimentos é aproximadamente a mesma, conforme Syswerda 1991 [Sys91] *apud* Schmitt e Amini [Sch98].

2.4.2.3.2.3 Mecanismos de seleção

Como já mencionado, o mecanismo de seleção deve utilizar um critério que privilegie os cromossomos (soluções) mais ajustados. Isso pode ser feito via um mecanismo conhecido como seleção por roleta viciada. Esse mecanismo é definido como:

Passo 1 Faça o somatório do nível de ajuste (fitness) de todos os cromossomos na população.

Passo 2 Gere um número aleatório entre 0 e o somatório dos níveis de ajuste dos cromossomos.

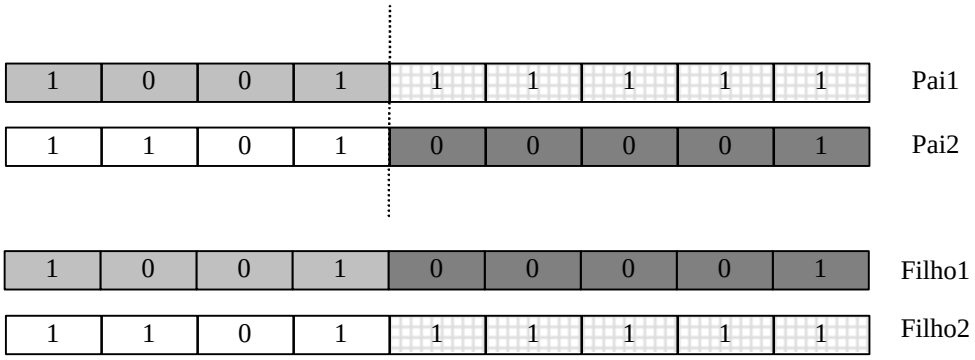


Figura III-5 : Crossover em 1 ponto

Passo 3 Selecione o primeiro cromossomo cuja soma acumulada dos fitness dos cromossomos anteriores seja maior ou igual ao número selecionado.

No caso do PCV, cada cromossomo representa uma rota. Assim, uma função natural para avaliar o fitness de cada um é o comprimento da rota. Como se trata de um problema de minimização, deve ser feito um ajuste a fim de que a seleção privilegie os cromossomos com probabilidade inversa ao valor do seu nível de ajuste, isto é, privilegiando rotas curtas e penalizando rotas longas.

Uma situação que pode ocorrer durante o processo de seleção é o aparecimento de cromossomos super ajustados em relação a todos os demais. Esse fato pode gerar uma convergência prematura do processo. Uma alternativa para contornar esse fenômeno, consiste na criação de um sistema de ponderação que dependa não do valor do ajustamento, mas da posição de um cromossomo em relação aos demais. De posse desse "rank", qualquer cromossomo que ocupe a primeira posição por exemplo, será sempre selecionado com a mesma probabilidade.

2.4.2.3.2.4 Crossover

O operador *crossover* é o mecanismo responsável pelo processo de combinação de cromossomos pais. Existem diversas maneiras, em particular no caso do PCV, de efetuarem-se essas combinações. O processo mais simples de cruzamento é conhecido como *crossover em um ponto*. Neste, tendo sido selecionados dois pais para o cruzamento, o processo consiste em escolher aleatoriamente uma posição entre 1 e L-1 (onde L é o comprimento dos cromossomos utilizados no problema). Cada um dos cromossomos selecionados é secionado na posição sorteada. Gera-se então um cromossomo concatenando o lado à direita da posição L de um dos pais, com o lado à esquerda da posição L do outro pai. Um segundo cromossomo é gerado através das outras duas partes restantes, conforme mostra a figura II-5.
Uma extensão desse mecanismo é o cruzamento com dois pontos de corte. Neste caso, são selecionados dois pontos de corte nos cromossomos pais e a informação contida entre esses pontos é intercambiada entre eles, gerando dois novos filhos.

O problema do PCV apresenta algumas dificuldades nesse processo. Com efeito, uma representação típica do problema, consiste na associação do número da cidade a cada gene do cromossomo, sendo a solução (a rota) dada pela seqüência em que tais genes aparecem no cromossomo. Existem ainda, outras formas de representação, por exemplo: através de adjacências ou por meio de matrizes. Estes casos não serão tratados aqui. Alguns exemplos dessas estruturas de representação podem ser vistos em Potvin [Pot96]. Considerando a representação por caminho, ao se aplicar o operador de *crossover* conforme acima, é possível que soluções inviáveis sejam geradas, figura II-6.
Diversos operadores de cruzamento têm sido propostos para o PCV a fim de contornar esse problema. A seguir, serão apresentados alguns desses operadores:

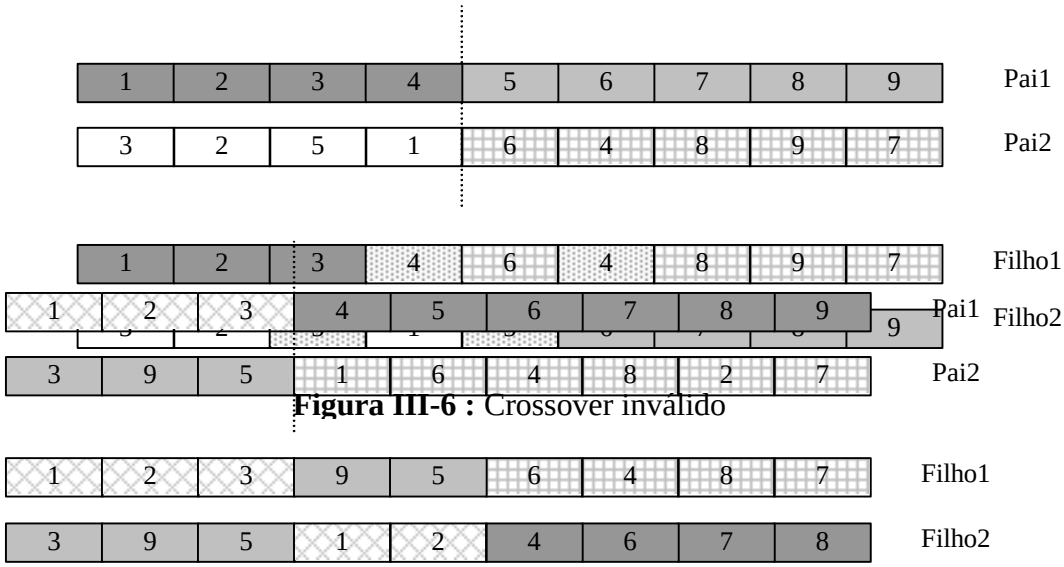


Figura III-6 : Crossover inválido

Figura III-7 : *MX- Modified Crossover*.

Crossover modificado MX- Modified crossover

Neste tipo de *crossover*, um ponto de corte é escolhido aleatoriamente em um dos cromossomos pai. O descendente é gerado primeiro adicionando-se os genes à esquerda do ponto de corte do primeiro pai, e em seguida, concatenando-se novos genes ao descendente na ordem com que aparecem no segundo pai, evitando-se a formação de soluções inviáveis. De modo análogo, o segundo filho é gerado, conforme figura II-7.

Crossover PMX (Partially Mapped Crossover)
Este operador tenta preservar a posição absoluta ocupada pelas cidades nos cromossomos pais. Para gerar os descendentes, são necessários os seguintes passos:

Passo 1 Dado dois cromossomos pais, p1 e p2, selecione aleatoriamente dois pontos de corte.

Passo 2 Gere um novo cromossomo f1 com o mesmo conteúdo do cromossomo p1 entre os pontos de corte e com as demais posições com o conteúdo de p2.

Passo 3 Identifique em f1 os pares de genes que estão gerando soluções inviáveis (valores duplicados no cromossomo).

Passo 4 Se o cromossomo é inviável, então existe um conjunto *A*, de pares de genes (a_i, a_i') com valores iguais. Seja a_i, o gene localizado entre os pontos de corte, e a_i' o externo a esses pontos (à direita ou à esquerda). Para todos os elementos de *A*, substitua o valor de cada gene a_i', pelo valor de p2 que cedeu lugar ao valor de a_i, quando o passo 2 foi executado. Volte ao passo 3.

Passo 5 Se os dois filhos já foram criados então pare. Caso contrário, execute os passos de 1 a 4 trocando p1 por p2, e f1 por f2, a fim de gerar o segundo descendente.

Veja o esquema na figura II-8.

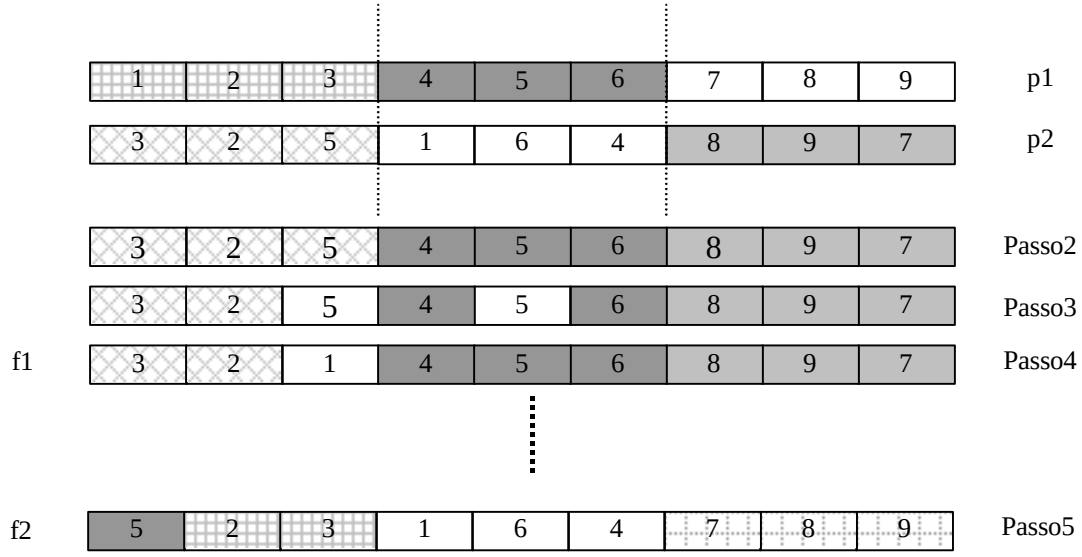


Figura III-8 : *PMX – Partial Mapped Crossover.*

Crossover OX - Order Crossover

Determinam-se de modo aleatório dois pontos de corte sobre os cromossomos pais. Cria-se um descendente, copiando-se para ele os elementos entre os pontos de corte, na posição em que aparecem em um dos pais. Em seguida, completa-se o descendente com os genes do outro pai e na ordem com que nele aparecem, partindo do segundo ponto de corte e evitando a formação de cromossomos inválidos. Procedimento análogo é utilizado para gerar o outro filho, figura II-9.

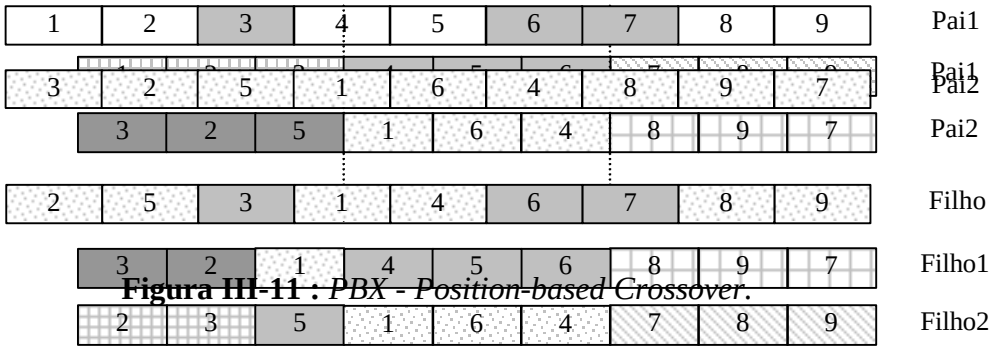


Figura III-11 : *PBX - Position-based Crossover.*

Figura III-9 : *OX – Order Crossover.*

Crossover OBX - Order Based Crossover
Escolhidos dois cromossomos pais, um primeiro pai é selecionado e dele são escolhidos *k* genes. Gera-se um descendente, substituindo os genes do pai escolhido nas posições ocupadas pelos genes de mesmo valor no outro pai, mas mantendo a mesma ordem que ocupavam no primeiro. A seguir, completa-se o descendente com os elementos do segundo cromossomo pai, evitando cromossomos inválidos, figura II-10.

Crossover PBX - Position-based *crossover*
Tendo sido selecionados dois cromossomos pais, um deles é escolhido e nele *k* posições são selecionadas. No descendente, esses genes são colocados na mesma posição e ordem que ocupam no pai escolhido. A seguir, completa-se o descendente com os elementos do outro pai, evitando-se a formação de cromossomos inválidos, figura II-11. Esse *crossover* pode ser visto como uma extensão do *crossover* OX.

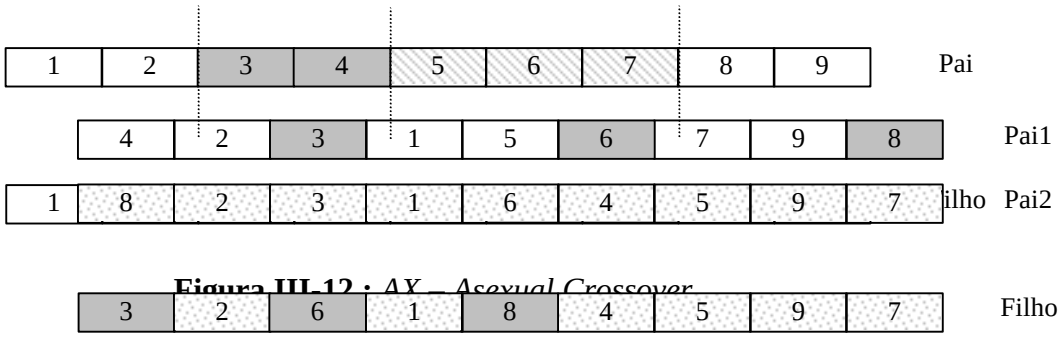


Figura III-12 : *AX – Asexual Crossover.*

Figura III-10 : *OBX - Order Based Crossover.*

Crossover AX – Asexual Crossover

Neste operador, um único cromossomo pai é utilizado na geração de um novo descendente. Isso é feito, selecionando-se três pontos de corte e trocando-se de posição os dois segmentos formados, figura II-12. Ao contrário dos demais, não apresenta a dificuldade de checagem de soluções inválidas. Esse fato garante grande velocidade na aplicação deste operador, Chatterjee *et al* [Cha96].

Além desses, muitos outros operadores de *crossover* têm sido propostos para a representação por caminho.

2.4.2.3.2.5 Mutação

A mutação em algoritmos genéticos tem como objetivo impedir uma prematura convergência da população.
Usualmente, o operador de mutação é um procedimento que seleciona aleatoriamente de posições em um cromossomo e modifica os valores contidos em tais posições. Por exemplo, no caso de representação em termos de 0's e 1's, uma vez selecionada uma posição que contém o valor 0, troca-se este valor por 1 e vice-versa. Tais taxas de troca utilizadas, em geral, apresentam valores em torno de 1/1000.
No caso do PCV, novamente algumas adaptações precisam ser feitas a fim de manter como resultante da mutação um cromossomo válido. Alguns mecanismos de troca são:
Swap (Permuta)
Selecionam-se aleatoriamente duas posições no cromossomo e as cidades contidas nessas posições têm sua posição trocada, conforme mostra a figura II-13.

Busca local

Após o *crossover*, os cromossomos filhos são submetidos a um processo de refinamento, utilizando-se uma heurística de busca local (*2-opt*, *3-opt*, etc).

Mistura

Nessa abordagem, a mutação se faz através da seleção aleatória de dois pontos de corte no cromossomo. No intervalo compreendido entre esses pontos, realiza-se uma mistura aleatória, executando-se uma série de swap's com os elementos aí contidos.

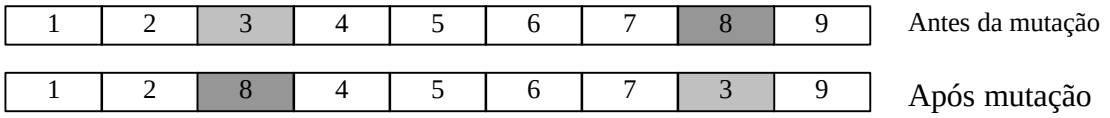


Figura III-13 : *Swap.*

Vários fatores influenciam na qualidade e eficiência das implementações de algoritmos genéticos para o PCV, entre eles: os mecanismos de geração de população, diferentes tipos de operadores de *crossover*, razão de mutação, etc. Segundo Potvin [Pot96], algumas afirmativas se enquadram em todas as situações:

a) a combinação de *crossover* e mutação busca local são críticas para o sucesso desses algoritmos.

b) uma maneira eficiente de tratar o problema está em dividir a população de cromossomos em sub-populações e através de processamento paralelo, buscar soluções para cada sub-população, realizando trocas periódicas de informações

genéticas entre elas. Isso evita convergências prematuras, pois mantém um bom nível de diversidade.

c) para qualquer implementação, a qualidade da solução final cresce com o aumento da população. Isto se verifica graças à diversidade encontrada em grandes populações; e

d) devem ser aplicados a problemas de porte médio, pois como usam muito tempo para resolver um problema, não são adequados para resolver problemas de grande porte.

Trabalho recente de Schmitt e Amini [Sch98] empregando 144 configurações diferentes de AG`s para o PCV, concluem que as configurações que geram soluções de melhor qualidade devem considerar: uma inicialização híbrida da população (sendo 50% gerada aleatoriamente e 50% gerada através da heurística de SFC descrita em Bartholdi e Paltzaman [Bar88]), uma grande população, uma estratégia de evolução em estado firme, e um operador de cruzamento AX ou OX.

2.4.2.3.3 Recozimento Simulado (Simulated Annealing)

Na física da matéria condensada, recozimento (annealing) é um processo térmico utilizado para obtenção de estados de baixa energia em um sólido. Esse processo consiste em duas etapas: na primeira, a temperatura do sólido é aumentada para um valor máximo no qual ele se funde; e na segunda, a temperatura é reduzida lentamente até que o material se solidifique.

Na segunda fase, o resfriamento deve ser realizado muito lentamente, possibilitando aos átomos que compõem o material, tempo suficiente para se organizarem em uma estrutura uniforme com energia mínima. Se o sólido for resfriado bruscamente, seus átomos formarão uma estrutura irregular e fraca, com alta energia em conseqüência do esforço interno gasto.

O recozimento pode ser visto como um processo estocástico de determinação da organização dos átomos em sólido que apresente energia mínima. Em altas temperaturas, os átomos se movem livremente, e com grande probabilidade podem mover-se para posições que incrementarão a energia total do sistema. Quando se baixa a temperatura, os átomos gradualmente se movem em direção a uma estrutura regular, e somente com pequena probabilidade incrementarão suas energias.

Esse processo foi simulado em computador, com sucesso, por Metropolis *et al* [Met53]. O algoritmo utilizado baseava-se em técnicas de Monte Carlo e gerava uma seqüência de estados de um sólido da seguinte maneira: dado um estado corrente *i* do sólido com energia E_i, um estado subseqüente era gerado pela aplicação de um mecanismo de perturbação, o qual transformava o estado corrente em um próximo estado por uma pequena distorção, por exemplo, pelo deslocamento de uma única partícula. A energia no próximo estágio passa a ser E_j. Se a diferença de energia $\Delta E = E_j - E_i$ fosse menor ou igual a zero, o estado *j* era aceito como estado corrente. Se a variação fosse maior que zero, o estado *j* era aceito com uma probabilidade igual a $e^{(-\Delta E / k_B T)}$ onde *T* é a temperatura e k_B é uma constante física conhecida como constante de Boltzmann. Essa regra de aceitação é conhecida como *critério de Metropolis*, e o algoritmo como *Algoritmo de Metropolis.*

Kirkpatrick *et al* [Kir83] desenvolveram um algoritmo de utilização genérica análogo ao de Metropolis, denominado *Algoritmo de Recozimento Simulado*. Nesse algoritmo, utilizaram como *critério aceitação* de uma nova solução, a função

$$P_{c_k}(aceitar\ j) = \begin{cases} 1 & se\ g(j) \le g(i) \\ exp\left(\dfrac{-[g(j) - g(i)]}{c_k}\right) & se\ g(j) > g(i) \end{cases} \qquad (16)$$

onde g é a função a ser otimizada (minimizada), *i* e *j* são a solução corrente e uma solução candidata, respectivamente e c_k um parâmetro denominado *temperatura*.

Segundo esse critério, se uma solução candidata *j* é melhor ($g(j) \le g(i)$) que a solução corrente *i*, esta é aceita com probabilidade 1. Caso contrário, a solução candidata é aceita com uma dada probabilidade. Essa probabilidade, que é dada pela equação (16), é maior na medida em que for menor *variação de energia*, definida por $\Delta = g(j) - g(i)$. Ao mesmo tempo, à medida que há um decréscimo da *temperatura* (c_k),

o algoritmo torna-se mais seletivo, passando a aceitar com menor freqüência ainda, soluções que apresentem grande aumento na variação de energia, isto é, soluções que sejam muito piores que a solução corrente. Essa probabilidade tende a zero à medida que a temperatura se aproxima do *ponto de congelamento*. O comportamento do critério de aceitação pode ser observado, para alguns valores, na tabela II-6.

Valores para $P_{C_K}(aceitar\ j)$ se g(j)>g(i) em função da variação de energia e da temperatura.

DEnergia= g(j)-g(i)	Temperatura (c_k)						
	0,0001	0,01	1	10	20	80	100
0,0001	0,3678	0,9900	0,9999	0,9999	1,0000	1,0000	1,0000
0,01	0,0000	0,3678	0,9900	0,9990	0,9995	0,9998	0,9999
1	0,0000	0,0000	0,3678	0,9048	0,9512	0,9875	0,9900
10	0,0000	0,0000	0,0000	0,3678	0,6065	0,8825	0,9048
20	0,0000	0,0000	0,0000	0,1353	0,3678	0,7788	0,8187
80	0,0000	0,0000	0,0000	0,0003	0,0183	0,3678	0,4493
100	0,0000	0,0000	0,0000	0,0000	0,0067	0,2865	0,3678

Tabela III-6

O algoritmo recozimento simulado pode ser descrito de modo genérico como:
Gere uma solução inicial *S* e ajuste a solução $S^*=S$.
Determine uma temperatura inicial $c=c_0$.

Enquanto *não atingir o ponto de congelamento* faça
    Enquanto *não alcançar o equilíbrio para esta temperatura* faça
        Escolha um vizinho aleatório *S'* da solução corrente.

        Faça $D=||S'||-||S||$.
        Se $D£0$ (movimento de melhora):
            Faça $S=S'$.
            Se $||S||<||S^*||$, faça $S^*=S$.
        Senão (movimento de diversificação):
            Gere um valor aleatório r~U[0,1][6]
            Se $r<e^{-D/c}$, faça $S=S'$.
    FimEnquanto
    Reduza a temperatura *c*.
FimEnquanto

Retorne S^*.

2.4.2.3.3.1 Programação de resfriamento

Esta programação diz respeito ao modo como o parâmetro de controle da temperatura *"c"* é manipulado durante a execução do algoritmo. Segundo Aarts *et al* [Aar97], uma programação de resfriamento consiste de um valor inicial para o parâmetro *"c"* e de uma seqüência finita de transições desses valores. Essas transições de "*c*" são regidas por uma função de rebaixamento da temperatura.
Muitas formas diferentes de programação de resfriamento têm sido propostas. De modo geral, estão dispostas em dois grupos: as programações estáticas e as dinâmicas. No primeiro grupo, estão as programações em que os parâmetros de resfriamento são fixos durante toda a execução do algoritmo, enquanto no segundo, tais parâmetros são adaptados durante a busca. Uma revisão desses mecanismos pode ser encontrada em Van Laahorven e Aarts [Van87] apud [Aar97].
Kirkpatrick *et al* [Kir83] introduziram uma programação de resfriamento que ficou conhecida como *programação geométrica*. Nessa abordagem o valor inicial do parâmetro de controle da temperatura é dado por um valor suficientemente grande $c_0=Dg_{max}$, onde Dg_{max} é a máxima diferença de custo entre duas

[6] Distribuição uniforme no intervalo [0,1].

soluções vizinhas. Em geral, utiliza-se uma estimativa desse valor. A temperatura é reduzida segundo a função $c_{k+1}=a.c_k$, $k=0, 1,...$, onde a é um valor constante positivo menor, porém, próximo de 1. Usualmente, esse valor pertence ao intervalo [0,8; 0,99]. O valor final do parâmetro de controle de temperatura é um valor pequeno fixo, que pode estar relacionado com a menor distância entre duas soluções vizinhas.
Nesse mesmo trabalho, aplicaram o recozimento simulado sobre problemas com 400 cidades, gerados aleatoriamente e contendo nove agrupamentos cada. Obtiveram resultados melhores, fixando a temperatura de congelamento em 0 (zero) e α=0,78939. Relataram também a aplicação desse mesmo esquema para um problema com 6000 cidades. Sem entrar em detalhes, afirmaram que os resultados alcançados foram "bons".

2.4.2.3.3.2 Aplicação ao PCV

Uma implementação básica para o PCV consiste na combinação de têmpera simulada com movimentos *2-opt*. Neste caso, soluções vizinhas são determinadas por meio de movimentos *2-opt* aleatórios, e sobre essas soluções o critério de aceitação é aplicado (RS_1). Em Johnson e McGeoch [Joh97], essa implementação foi comparada com procedimentos *2-opt*, *3-opt* e *LK*, sobre problemas com 100, 316 e 1000 pontos dispostos aleatoriamente em um plano e considerando a distância euclidiana. Utilizaram como temperatura inicial o valor *n.(n-1)* e como critério de congelamento, uma seqüência de 5 temperaturas sem redução de custo e com percentual de aceite inferior a 2%. Os resultados são sumarizados na tabela II-7.

Desempenho do recozimento simulado usando vizinhança 2-opt completa e alta temperatura inicial (RS_1): problemas com distâncias *euclidianas.*

	Tempo médio de execução (segundos)			*Excesso médio para o limite de Held/Karp*		
n	*100*	*316*	*1.000*	*100*	*316*	*1.000*
RS_1 *(básico)*	12,40	188,00	3.170,00	5,20	4,10	4,10
RS_1*+2-opt*	-	-	-	3,40	3,70	4,00
2-opt	0,03	0,09	0,34	4,50	4,80	4,90
3-opt	0,04	0,11	0,41	2,50	2,50	3,10
LK	0,06	0,20	0,77	1,50	1,70	2,00

** Adaptada de Johnson e McGeoch [Joh97]*

Tabela III-7

Como se pode notar, os resultados gerados por RS_1 são similares aos obtidos com *2-opt* e *3-opt*, embora, o tempo necessário para alcançá-los, sejam muito maiores. É notável também a vantagem, tanto em tempo quanto em qualidade obtida pelo algoritmo *LK*.
Relatam também, que aumentando o tempo de permanência em cada temperatura em 10 vezes, para problemas com n=100 cidades, obtiveram resultados mais precisos que os obtidos por 3-opt (3,4% contra 1,9% de excesso médio). Em contrapartida, esse ganho implicou em um aumento do tempo de execução em cerca de 3.000 vezes. E ainda, que aumentando esse tempo de permanência em cada nível de temperatura em 100 vezes, em problemas com n=1.000 cidades, os resultados obtidos com RS_1 foram similares aos alcançados com *LK*. Contudo, essa estratégia produziu um aumento no tempo de processamento em cerca de 20.000 vezes.
Entre as estratégias que podem ser utilizadas para acelerar o recozimento simulado, estão a determinação de vizinhanças mais restritas e a utilização de programações de resfriamento que iniciem com baixas temperaturas.
Bonomi e Lutton [Bon84] propuseram que a vizinhança fosse construída como se segue: para vértices distribuídos no plano e distâncias euclidianas, determina-se o menor quadrado capaz de envolver todos os pontos, e em seguida particiona-se este em m^2 regiões. Observando que, no caso de trocas *2-opt*, a escolha de dois arcos (t_1, t_2) e (t_3, t_4) para exclusão implica na determinação de dois outros arcos para inclusão (t_1, t_4) e (t_2, t_3) que formem um circuito, limita-se à escolha do vértice t_3 a aqueles que pertencem à mesma região de t_2 ou a regiões que fazem fronteiras com ela.
Quanto à programação de resfriamento, é possível utilizar alguma heurística de construção de circuitos para obter um circuito inicial. Sobre essa solução inicial, utiliza-se o recozimento simulado como

mecanismo de aprimoramento da solução. Com isso, a temperatura inicial pode ser ajustada para valores mais baixos, provocando uma redução no tempo total de busca. Um comparativo de uma implementação, utilizando vizinhanças e baixas temperaturas (RS_2) com RS_1, *2-opt*, *3-opt*, *LK*, para problemas euclidianos gerados aleatoriamente no plano com 100, 316 e 1.000 cidades, pode ser visto na tabela II-8.

Desempenho de variantes do recozimento simulado, sendo mantida $\alpha n(n-1)$ vezes a cada temperatura (condição de equilíbrio), comparadas com execuções do *2-opt*, *3-opt* e LK para problemas gerados aleatoriamente com distâncias *euclidianas*.

		% médio de excesso			*Tempo médio de execução (segundos)*		
Variante	α	*100*	*316*	*1.000*	*100*	*316*	*1.000*
RS_1 (básico)	1	3,4	3,7	4,0	12,40	188,00	3.170,00
RS_1 + vizinhança	1	2,7	3,2	3,8	3,20	18,00	81,00
RS_2 (baixa temp. inicial + vizinhança)	10	1,6	1,8	2,0	14,30	50,30	229,00
2-opt		4,5	4,8	4,9	0,03	0,09	0,34
Melhor solução de 10.000 repet. indep. de 2-opt		1,7	2,6	3,4	66,00	161,00	517,00
3-opt		2,5	2,5	3,1	0,04	0,11	0,41
Melhor solução de 10.000 repet. indep. de 3-opt		0,9	1,2	1,9	113,00	326,00	1.040,00
LK		1,5	1,7	2,0	0,06	0,20	0,77
Melhor solução de 100 repet. indep. de LK		0,9	1,0	1,4	4,10	14,50	48,00

** Adaptada de Johnson e McGeoch [Joh97]*

Tabela III-8

Como se observa a utilização conjunta de vizinhanças, heurísticas para a construção de rotas iniciais e um maior tempo de permanência em cada temperatura, provoca uma grande redução no tempo de execução e uma sensível melhora da qualidade das soluções obtidas se comparada à versão básica do algoritmo (RS_1). Essas soluções apresentam ainda valores comparáveis aos produzidos pelo algoritmo *LK*, porém, com desvantagens no item tempo de execução.

2.4.2.4 Outros métodos

2.4.2.4.1 GENIUS

O procedimento GENIUS foi proposto por Gendreau *et al* [Gen92]. É composto por duas etapas. A primeira denominada *Generalizaded Insertion Procedure* (*GENI*), é responsável pela construção de uma rota inicial a exemplo de outros procedimentos já apresentados (seção 2.4.2.2). A segunda fase denominada *Unstringing and Stringing* (*US*) é responsável pela melhoria da rota construída pelo procedimento *GENI*.

A principal característica de *GENI* é que quando da inserção de um vértice *v*, este toma lugar, não necessariamente entre dois vértices consecutivos na rota. Embora após a inserção estes passem a ser adjacentes a *v*.

Para cada vértice *v* a ser adicionado à rota, o procedimento considera dois tipos de inserção, para cada uma das duas orientações da rota.

- *Inserção tipo I*: Seja v_k-v_l e v_k-v_j. A inserção de *v* na rota, resulta na remoção dos arcos (v_i, v_{i+1}), (v_j, v_{j+1}) e (v_k, v_{k+1}) e na inserção dos arcos (v_i, v), (v, v_j), (v_{i+1}, v_k) e (v_{j+1}, v_{k+1}) na rota.

Pode-se notar, que no caso particular $j=i+1$ e $k=j+1$ a inserção tipo I, torna-se um procedimento de inserção padrão.

- *Inserção tipo II*: Seja v_k¹v_j e v_k¹$v_{j+1;}$ v_l¹v_i e v_l¹v_{i+1}. A inserção de v na rota, resulta na remoção dos arcos (v_i, v_{i+1}), (v_{l-1}, v_l), (v_j, v_{j+1}) e (v_{k-1}, v_k), e na inserção dos arcos (v_i, v), (v, v_j), (v_l, v_{j+1}), (v_{k-1}, v_{l-1}) e (v_{i+1}, v_k) na rota.

O número de escolhas potenciais de v_i, v_l, v_j e v_k é da ordem n^4. Esse número é limitado para um vértice v, pela criação de um conjunto de cardinalidade p, $N_p(v)$, definido como o conjunto dos p pontos mais próximos de v, que já estão na rota. Primeiro seleciona-se v_i, v_j Î $N_p(v)$, e então v_k Î N_p (v_{i+1}) e v_l Î N_p (v_{j+1}). Consideram-se, também, todas as inserções de v entre dois vértices consecutivos v_i e v_{i+1}, com v_i Î $N_p(v)$. Utilizaram em seus testes valores para p variando de 2 a 7. Esquemas representativos das inserções tipos I e II podem ser vistos na figura II-14.

O algoritmo do procedimento *GENI* é descrito como:

Passo 1 — Gere uma rota inicial qualquer com 3 vértices. Em seguida inicialize todos os $N_p(v)$ para todos os vértices fora da rota.

Passo 2 — Selecione um vértice qualquer fora da rota e avalie as duas inserções para ambos os sentidos na rota. Execute a inserção que apresentar melhor resultado (inserção de menor custo). Atualize as p-vizinhanças para os vértices fora da rota, considerando que v agora está na rota.

Passo 3 — Repita o passo 2 até não haver mais pontos fora da rota.

O procedimento *US* é um algoritmo de pós-otimização, que conta com um mecanismo de retirada e outro de reinserção de um vértice em uma rota viável. O procedimento de inserção do vértice é o mesmo do *GENI*. O de retirada é definido como:

- Unstringing tipo I: Seja v_j Î N_p (v_{i+1}), e para uma dada orientação seja v_k Î N_p (v_{i-1}) um vértice no caminho (v_{i+1},..., v_{j-1}). Então os arcos ($v_{i-1,}$ v_i), ($v_{i,}$ v_{i+1}), ($v_{k,}$ v_{k+1}) e (v_j, v_{j+1}) são removidos, enquanto os arcos ($v_{i-1,}$ v_k), ($v_{i+1,}$ v_j) e ($v_{k+1,}$ v_{j+1}) são inseridos. Os dois caminhos (v_{i+1},..., v_k) e (v_{k+1},..., v_j) são invertidos.
- Unstringing tipo II: Seja v_j Î N_p (v_{i+1}), e para uma dada orientação seja v_k Î N_p (v_{i-1}) um vértice no caminho (v_{j+1},..., v_{i-2}), seja também v_l Î N_p (v_{k+1}) um vértice no caminho (v_j,..., v_{k-1}). Então os arcos ($v_{i-1,}$ v_i), ($v_{i,}$ v_{i+1}), ($v_{j-1,}$ v_j), (v_l, v_{l+1}) e (v_k, v_{k+1}) são removidos, enquanto os arcos ($v_{i-1,}$ v_k), ($v_{l+1,}$ v_{j-1}), (v_{i+1}, v_j) e ($v_{l,}$ v_{k+1}) são inseridos. Os dois caminhos (v_{i+1},..., v_{j-1}) e (v_{l+1},..., v_k) são invertidos.

O algoritmo para utilização de *US* é definido como:

Passo 1 — Considere uma rota inicial t de custo z. Faça $t^*=t$, $z^*=z$ e $t=1$.

Passo 2 — Partindo de t, aplique os procedimento de remoção e inserção do vértice v_t, considerando em cada caso ambas operações e as duas orientações possíveis de rota. Faça t' ser a rota obtida e z' o melhor custo. Faça $t=t'$ e $z=z'$.

Passo 3 — Se $z<z^*$, faça $t^*=t$, $z^*=z$ e $t=1$. Repita o passo 2.

Passo 4 — Se $z \geq z^*$, faça $t=t+1$.

Passo 5 — Se $t=n+1$ pare: a melhor rota obtida solução é t^* e tem custo z^*. Caso contrário repita volte para o passo 2.

Gendreau *et al* [Gen92] compararam os resultados obtidos pela utilização de GENIUS, contra o algoritmo CCAO [Gol85]. Para problemas gerados aleatoriamente no plano, em instâncias com tamanho variando de 100 a 500 pontos, os resultados obtidos apresentaram ganho na qualidade na ordem de 2% em favor do procedimento GENIUS, nos casos mais favoráveis. Testaram também esse algoritmo sobre problemas conhecidos, com dimensões variando de 100 a 532 pontos. Obtiveram resultados comparáveis aos conseguidos por outras técnicas, como recozimento simulado e busca tabu.

2.5. Considerações finais

Além dos mecanismos de resolução aqui apresentados, muitos outros são encontrados na literatura, utilizando heurísticas na resolução do PCV, como: *Ant Path, Redes Neurais Artificiais, Tabu Search* entre outras. Fazendo um apanhado geral dos procedimentos discutidos no decorrer desse capítulo, constata-se que:

a) os métodos exatos têm méritos em relação a qualidade dos resultados que são capazes de obter.

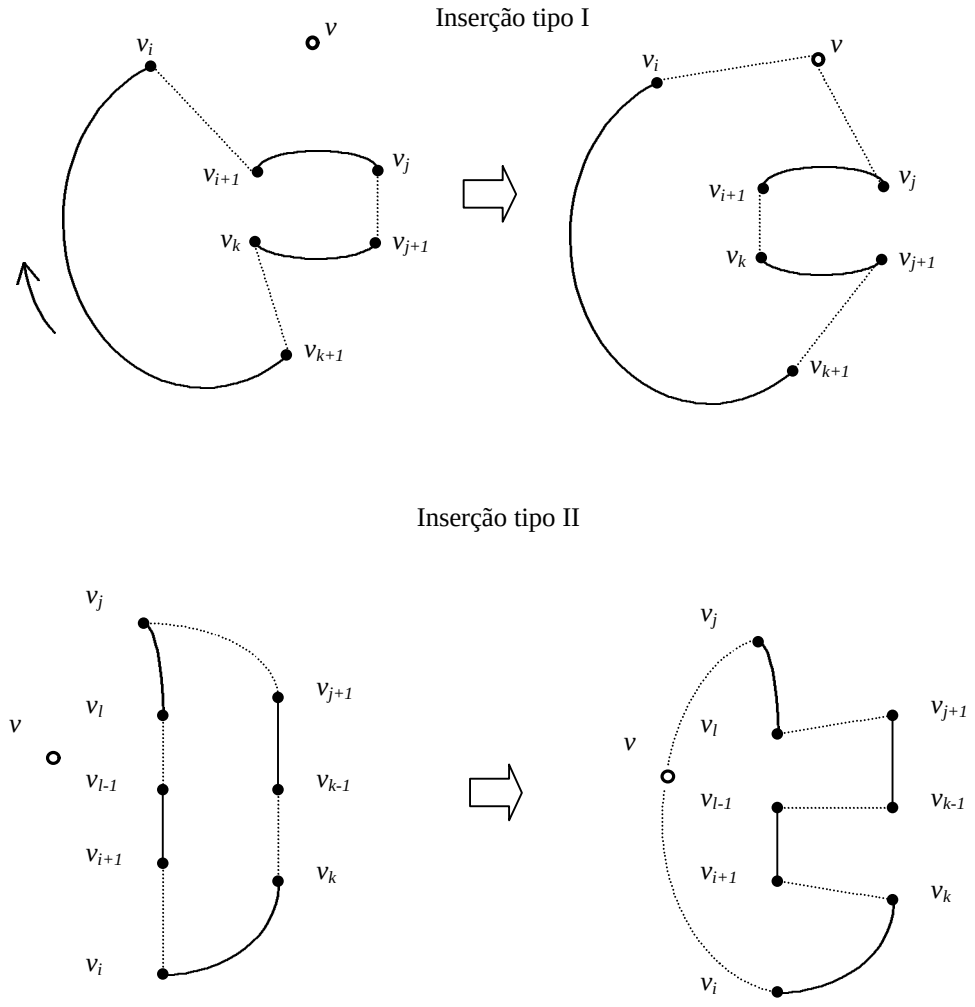


Figura III-14 : Inserções Tipos I e II do algoritmo *GENIUS.*

Por outro lado, é inegável a quantidade de tempo e de recursos computacionais que necessitam para isso. Têm ainda contra si, a complexidade de programação dos algoritmos, principalmente em suas versões mais sofisticadas. Portanto, na falta de hardware poderoso e de tempo, parece limitado, hoje, a resolução de problemas não muito grandes;

b) os procedimentos de construção de rotas são velozes, fáceis de programar e exigem poucos recursos de hardware, em geral. Porém, não fazem frente a resultados obtidos por outros métodos de busca local. Freqüentemente, são utilizados como mecanismo de aceleração destes.

A idéia é que, partindo de uma solução não muito ruim e obtida rapidamente, tais métodos economizam tempo, evitando buscas em regiões menos promissoras do espaço de soluções. Esse princípio tem sido utilizado para acelerar procedimentos como recozimento simulado, algoritmos genéticos e procedimentos de busca *k-opt*, por exemplo. Entretanto, esta estratégia não parece ser crucial em uma heurística mais moderna como a *Busca Local Dirigida*, a ser discutida no capítulo III, que não utiliza essa estratégia e obtêm bons resultados assim mesmo;

c) a utilização de algoritmos genéticos no tratamento do PCV é limitada a problemas com não mais que poucas centenas de vértices. Em sua variante mais promissora, denominada *Busca Local Genética (Genetic Local Search)*, que utiliza uma combinação de procedimentos de melhoria de rota, juntamente com heurísticas de construção de rotas inicias, conforme descrito por Freisleben e Merz [Frei96a] e [Frei96b], os resultados obtidos apesar de bons, exigem muito tempo de computação. Portanto, não devem ser a primeira escolha no tratamento de problemas com grandes dimensões, caso o fator tempo for importante;
d) o recozimento simulado é citado freqüentemente com uma poderosa metaheurística. É capaz de alcançar bons resultados desde que tempo suficiente seja fornecido a ele. É muito fácil de programar;
e) em 1994 o procedimento *GENIUS* foi utilizado em conjunto com a busca tabu na resolução do problema do roteamento de veículos, por Gendreau *et al* [Gen94]. Os resultados apresentados por essa combinação mostraram-se bastante competitivos;
f) os algoritmos de troca *k-opt* são certamente os mais utilizados, não somente em seu estado puro, mas também em combinação com outras técnicas como as metaheurísticas recozimento simulado, algoritmos genéticos, busca local dirigida, entre outras. Dentre os procedimentos *k-opt*, o *LK-iterado* é considerado, com justiça, um dos procedimentos mais poderosos entre os métodos heurísticos para resolução do PCV na atualidade.

Capítulo IV

3. Busca Local Dirigida

3.1. Considerações iniciais

Neste capítulo serão apresentados os elementos associados à metaheurística denominada *Busca Local Dirigida* (*BLD*), introduzida por Voudouris [Vou97] e Voudouris e Tsang [Vou99]. Adicionalmente, será descrito um mecanismo para ajuste dos parâmetros desse algoritmo.

3.2. Busca Local Dirigida (BLD)

Seja um problema de otimização combinatorial, onde *S é o subconjunto de soluções* e g a *função objetivo* associada. A *BLD* tenta otimizar *g*, utilizando uma seqüência de funções h_j geradas iterativamente através da agregação de termos à função *g*. Sobre cada função h_j, um procedimento de busca local é aplicado. Encontrado um mínimo local, os termos são modificados e a busca é reiniciada, prosseguindo nesse processo até que um critério de parada seja alcançado. Cada função h_j construída funciona como um mecanismo de direcionamento da busca local, tentando orientá-la para regiões mais promissoras no espaço de soluções. A seguir, a descrição dos elementos que compõe a *BLD*.

Um primeiro elemento da *BLD* é chamado *característica de uma solução*. Denotada por f_i, é definida como qualquer propriedade da solução que satisfaz uma restrição simples e que não seja trivial[7]. As *características de uma solução*, ou simplesmente *características*, são dependentes do problema e servem como uma interface entre a metaheurística e uma aplicação em particular. Restrições às *características* são introduzidas ou reforçadas com a base de informações sobre o problema e também durante o curso da busca local.

A cada característica está associado um valor, denominado *custo da característica*, que representa o impacto da correspondente propriedade da solução, sobre o custo da solução. O custo da característica pode ser constante ou variável.

Uma função *I* é utilizada para representar cada *característica* f_i, e é denominada *função indicadora da característica* definida como

$$I_i(s) = \begin{cases} 1 & se\ a\ solução\ s\ tem\ a\ propriedade\ f_i \\ 0 & caso\ contrário \end{cases} \qquad ,s \in S\ .$$

Restrições sobre as características tornam-se possíveis pelo aumento da *função de custo g* (função objetivo) do problema, através da inclusão de um conjunto de termos de penalidade. A nova função custo formada é denominada *função de custo aumentada* e é definida como:

$$h(s) = g(s) + l.\sum_{i=1}^{M} p_i.I_i(s), \qquad (1)$$

onde *M* é o *número de características* definidas sobre as soluções, p_i os *parâmetros de penalidade* correspondentes às características f_i e l um parâmetro de *controle da intensidade das restrições* com respeito ao custo da solução atual. Deve-se notar que *h* será modificada durante o processo de busca, gerando assim uma seqüência de funções h_j. Por simplicidade, será omitida a utilização de um índice para *h*.

O parâmetro de penalidade p_i dá o grau de restrição à *característica da solução* f_i.

O parâmetro l representa a importância relativa das penalidades, com respeito ao custo da solução. Provendo assim, um meio para controlar a influência dessa informação sobre o processo de busca (quanto maior o l, maior a importância das características e das penalidades). A *BLD*, iterativamente, repassa a

[7] "Não ser trivial" significa que: nem todas as soluções gozam dessa mesma propriedade.

função de custo aumentada para o procedimento de busca local, simplesmente, modificando o *vetor de penalidades*

$$p=(p_1,...,p_M)$$

cada vez que um mínimo local é encontrado para *h*.
Inicialmente, todos os parâmetros de penalidade são ajustados para *0* (isto é, não há características penalizadas). Em seguida, a função *de custo aumentada, h,* é repassada para o procedimento de busca local. Após o primeiro mínimo local e depois, sempre que um novo mínimo é alcançado, o algoritmo faz uma ação de modificação sobre a *função de custo aumentada*. A ação de modificação é feita incrementando em 1 (um) o parâmetro de penalidade de uma ou mais características presentes no mínimo local. O procedimento é finalizado quando um critério de parada (tempo, número iterações, etc.) é alcançado.
Informações prévias e históricas são gradualmente utilizadas para guiar o processo de busca local, à medida que os parâmetros de penalidade são incrementados.
Para cada característica f_i, definida sobre as soluções, está associado um custo c_i (considere tais custos constantes durante todo o processo de busca), representado pelo vetor

$$c=(c_1,...,c_M);\ c_i \geq 0\ "\ i.$$

Deve-se observar que as soluções de mínimo local encontradas durante o processo de busca local, dizem respeito à *função de custo aumentada*, e, portanto, tais soluções podem não ser soluções de mínimo local, com respeito à *função de custo* do problema, *g*. Antes das penalidades serem aplicadas as funções *g* e *h* são idênticas, mas com o processo de busca e as modificações decorrentes do ajuste do vetor de penalidades, isso deixa de ser verdade.
Para escapar dos mínimos locais, a *BLD* usa um *mecanismo de modificação de penalidades*. Este é responsável pela manipulação da *função de custo aumentada*, quando um mínimo local é atingido. Uma solução de mínimo local *s** em particular exibe um certo número de características. Então, os indicadores dessas características f_i assumem o valor 1 (isto é, $I_i(s^*)=1$). Quando um mínimo local é alcançado, os parâmetros de penalidades são incrementados em 1 para todas as características que maximizam a *função de utilidade* dada por

$$Util(s^*,f_i)=I_i(s^*).c_i / (1 + p_i). \qquad (2)$$

O parâmetro de penalidade, p_i, é incorporado na equação (2) para prevenir que o esquema seja totalmente viciado, com respeito à penalização de características com alto custo. Se uma característica *i* é penalizada muitas vezes, então $c_i / (1+p_i)$ na equação (2) decresce, o que permite diversificar escolhas e dá chance a outras características também serem penalizadas. A política implementada é que características são penalizadas com freqüência proporcional ao seu custo. Assim, características de alto custo são penalizadas mais freqüentemente.
Nota-se também, que é necessário calcular a utilidade somente para características presentes no mínimo local, pois características não presentes têm utilidade igual a zero.
Dependendo da força das penalidades na equação (1), uma ou mais iterações de modificação de penalidade, como descrito, são requeridas antes que o movimento seja feito para fora do mínimo local. Altos valores de l, fazem com que o algoritmo mais rapidamente escape de um mínimo local encontrado, enquanto baixos valores de l fazem o algoritmo mais cuidadoso, requerendo mais acréscimos nas penalidades antes de escapar. Baixos valores, embora tornem a fuga de um mínimo local mais lenta, conduzem a uma exploração mais cuidadosa do espaço de busca, pois reduzem a importância das penalidades na *função de custo aumentada, h(s).*
O algoritmo básico para *BLD* pode ser descrito como:

Procedimento BLD(*S, g, l*, [I_1,..., I_M], [c_1,..., c_M], *M*)
Início
k←0;
s_0←solução inicial gerada por alguma heurística ou aleatoriamente;
Para *i*←*1* até *M* **faça** p_i←*0;*
Enquanto *critério de parada* **faça**

$$h \leftarrow g + l \sum_{i=1}^{M} p_i .I_i \ ;$$

s_{k+1}←*BuscaLocal*(s_k, *h*);
Para *i*←*1* até *M* **faça** $util_i \neg I_i(s_{k+1}).c_i/(1+p_i);$
Para cada *i* tal que $util_i$ é máximo **faça** $p_i \leftarrow p_i+1$;
k←*k+1*;
FimEnquanto;
s^*←a melhor solução encontrada com respeito à função custo g;

Retorne s^*;
Fim.

A aplicação do algoritmo *BLD* para um problema qualquer, usualmente envolve: a definição das características que serão usadas, o assinalamento dos custos associados a cada característica e finalmente, a substituição do *procedimento BuscaLocal(),* presente no procedimento *BLD*, por um algoritmo de busca local para o problema em questão.

3.3. Busca Local Rápida (*BLR*)

No algoritmo de *BLD* há uma chamada ao procedimento *BuscaLocal()*. Para ocupar o lugar desse procedimento, um algoritmo de busca local adequado ao problema que se está tratando deve ser utilizado. No trabalho de Voudouris e Tsang [Vou99], onde esse procedimento é utilizado especificamente sobre o PCV, é utilizado um procedimento denominado *Busca Local Rápida* (*Fast Local Search)* em conjunto com *BLD*. De tal combinação, resulta um algoritmo que apresenta performance superior na resolução do PCV, se comparado a outras metaheurísticas como busca tabu, tempera simulada, e até certo ponto, a algoritmos especializados como *LK-iterado.*

O procedimento *BLR* trabalha com a idéia de redefinição dinâmica da vizinhança de uma solução. Isso é feito através da subdivisão da mesma, em partes menores e pela adoção de um critério de permissão ou não da pesquisa a cada uma dessas partes. Como resultado, essa conduta produz uma redução do tempo de busca em função da modificação (redução) dinâmica da vizinhança a ser pesquisada.

Explicando melhor, considere uma solução *s.* A *BLR,* toma a vizinhança de *s,* particiona-a em um número de pequenas sub-vizinhanças e associa um bit de ativação a cada uma das partes. Passa então, a examinar continuamente cada uma das partes em uma ordem preestabelecida, buscando somente naquelas em que o bit de ativação é igual a 1. Estas partes são denominadas *sub-vizinhanças ativas*. As demais, com bits iguais a zero, são chamadas *sub-vizinhanças inativas* e nelas não é feita busca. O processo de busca não é reiniciado toda vez que uma solução melhor é encontrada, em vez disso continua com a próxima sub-vizinhança ativa.

Inicialmente, todas as sub-vizinhanças são ativadas. Se uma sub-vizinhança é examinada e não contém um movimento de melhora, então ela é desativada. Caso contrário ela permanece ativa e o movimento de melhora encontrado é realizado. Dependendo do movimento executado, um número de outras sub-vizinhanças também é ativado. Em particular, ativam-se todas as sub-vizinhanças onde se espera que ocorram outros movimentos de melhora como resultado de um movimento recém executado. Com a melhora da solução, o processo extingui-se com menos e menos sub-vizinhanças sendo ativadas, até que todos os bits das sub-vizinhanças se tornem zero. A melhor solução encontrada até este ponto é retornada como um mínimo local aproximado.

A combinação de *BLR* com *BLD* é direta. A idéia chave está em associar o item *características de uma solução* da *BLD* a sub-vizinhanças da *BLR*.

A combinação de *BLD* com a *BLR* é feita no pseudocódigo a seguir:

Procedimento BLD_BLR(S, g, l, [I_1,..., I_M], [c_1,..., c_M], M, L)
Início
    $k \leftarrow 0$;
    $s_0 \leftarrow$ solução inicial gerada por alguma heurística ou aleatoriamente;
    Para $i \leftarrow 1$ **até** M **faça** $p_i \leftarrow 0$; *{Faz todas as penalidades iguais a zero}*
    Para $i \leftarrow 1$ **até** L **faça** $bit_i \leftarrow 1$; *{Ativa todas as sub-vizinhanças}*
    Enquanto *critério de parada* **faça**
        $h \leftarrow g + l \sum_{i=1}^{M} p_i . I_i$;
        $s_{k+1} \leftarrow$ BLR(s_k, h, [bit_1,...,bit_L], L);
        Para $i \leftarrow 1$ **até** M **faça** $util_i = I_i(s_{k+1}).c_i/(1+p_i)$;
        Para cada i tal que $util_i$ é máximo **faça**
            $p_i \leftarrow p_i + 1$;
            SetBits$\leftarrow$**SubVizinhançasCaracterística**(i);
            {Ativa todas as sub-vizinhanças relativas à característica penalizada}
            Para cada bit b **em** SetBits **faça** $b \leftarrow 1$;
        FimPara;
        $k \leftarrow k+1$;
    FimEnquanto;

s^*←a melhor solução encontrada com respeito à função custo g;
Retorne s^*;
Fim
Procedimento BLR (*S, h,* [bit_1,..., bit_L], *L*)

Início

Enquanto ∃*bit, bit=1* **faça**
Para *i←1* **até** *L* **faça**
Se bit_i*=1* **então** *{Busca de sub-vizinhanças para mov. de melhora}*
Mov←conjunto de movimentos na sub-vizinhança de i
Para cada movimento *m* em *Mov* **faça**
s'←m(s); {s' é a solução gerada pelo o movimento m}
Se *h(s')<h(s)* **então**
bit_i*←1*;
SetBits←**SubVizinhançasParaMovimento**(m);
{Ativação de outras sub-vizinhanças}
Para cada bit *b* **em** *SetBits* **faça** *b←1*;
s←s';
Vá para MovimentoDeMelhoraEncontrada;
FimSe;
FimPara;
{Desativa uma sub-vizinhança}
bit_i*←0;*
FimSe;
MovimentoDeMelhoraEncontrada: **Continue;**
FimPara*;*
FimEnquanto;
Retorna *s*;
Fim.
SubVizinhançasParaMovimento *(m)*
Procedimento que retorna os bits da sub-vizinhança para propagação da ativação quando o movimento *m* é realizado;
SubVizinhançasCaracterística (*i*)
Procedimento que retorna os bits da sub-vizinhança correspondentes a característica *i*;

Fazendo uma leitura do algoritmo, vê-se que inicialmente todos os bits de ativação são ajustados para 1 e é permitido a *BLR* encontrar o primeiro mínimo local. Depois, e sempre que uma característica é penalizada, os bits associados às sub-vizinhanças penalizadas são ajustados para 1. Desta maneira, após o primeiro mínimo local ser encontrado, a chamada de *BLR* examinará, em princípio, somente um número restrito de sub-vizinhanças, em particular aquelas as quais se associam características pouco penalizadas. Com isso, a busca local se concentra na remoção das características penalizadas da solução, ao invés de considerar todas as possíveis modificações.

3.4. Aplicação da BLD combinada com BLR ao PCV

Conforme já discutido no capítulo II, os procedimentos de busca local *2-opt*, *3-opt* e *LK* melhoram uma rota, trocando repetidamente arcos em um trajeto corrente por outros dois (*2-opt*), três (*3-opt*) ou mais (*LK*) outros arcos que não estejam no percurso.
Nesses procedimentos, cada vértice é visitado na ordem do trajeto, sendo cada um ou ambos os arcos adjacentes, examinados quanto a uma possível troca que melhore a solução corrente.
Na aplicação da *BLD* com a *BLR* o procedimento é semelhante. Inicialmente, todas as sub-vizinhanças estão ativas. Cada sub-vizinhança é examinada em uma ordem arbitrária estática (por exemplo, da primeira a *n*-ésima cidade). Cada vez que uma sub-vizinhança ativa é encontrada, tenta-se encontrar um movimento de melhora *2-opt, 3-opt* ou *LK* (de acordo com a heurística usada) a envolvendo. Se não for encontrado um movimento, a sub-vizinhança é feita inativa (o bit correspondente é ajustado para 0). Caso contrário, o primeiro movimento encontrado é realizado e as sub-vizinhanças correspondentes às cidades no fim de cada arco envolvido (removido ou adicionado pelo movimento) são ativadas (os bits correspondentes são ajustados para 1). O processo sempre continua com a próxima sub-vizinhança na

ordem estática. Sempre que uma volta completa por todas as sub-vizinhanças ocorra, sem que qualquer movimento de melhora seja encontrado, o processo termina e retorna o trajeto encontrado.
Completando a descrição dos elementos necessários a aplicação da *BLD*, definem-se o conjunto de *características de uma solução* como todos os arcos e_{ij} do grafo que representa o PCV; o *custo da característica* como o comprimento de cada arco, d_{ij}; e a função *indicadora da característica* por

$$Util(trajeto, e_{ij}) = Ie_{ij}(trajeto).\ d_{ij}/(1 + p_{ij})$$

onde

$$Ie_{ij}(trajeto) = \begin{cases} 1\ ; & se\ e_{ij}\ pertence\ ao\ trajeto \\ 0\ ; & caso\ contrário \end{cases}.$$

Das 24 diferentes combinações de procedimentos implementados por Voudouris [Vou97] para o PCV, ele avalia que a combinação de melhor performance é *BLD* com *BLR* e *2-opt*. Por essa razão, essa será a combinação escolhida para compor o algoritmo a ser descrito no capítulo IV.

3.5. Ajuste de l

Para executar a *BLD* é preciso especificar o valor do parâmetro de *controle da intensidade das restrições* λ. Voudouris e Tsang [Vou99] sugerem que tal valor deva ser calculado através da função

$$l = a.\frac{g(mínimo\ local\ na\ primeira\ iteração)}{n}. \qquad (3)$$

Como se vê, isso acaba por introduzir um novo parâmetro *"a"* que deve também ser especificado. Segundo o mesmo autor, o valor de "*a"* depende do tipo de algoritmo utilizado no procedimento de busca local. Sugere ainda, limites para esses valores conforme tabela III-1.

Intervalos sugeridos para valores de *"a"* quando utilizando a BLD combinada com diferentes heurísticas para o PCV.

Heurística	*Intervalo sugerido para a*
2-opt	$1/8 \le a \le 1/2$
3-opt	$1/10 \le a \le 1/4$
LK	$1/12 \le a \le 1/6$

Tabela IV-1

3.6. Calibração de *"a"* e determinação do critério de parada.

Em muitos algoritmos heurísticos, como alguns dos discutidos no capítulo II, por exemplo, a escolha do valor de parâmetros e de critérios de parada ficam quase sempre sob responsabilidade do interessado. Nesse aspecto a *BLD* não é uma exceção.
Isso é um inconveniente da utilização desse tipo de algoritmo, particularmente em procedimentos automáticos, como o que será visto no capítulo IV.
Portanto, a busca de um critério automático de escolha do valor parâmetro *"a"* que produza uma convergência do algoritmo no menor tempo (número de iterações) possível, é necessária. Além disso, e com o mesmo objetivo, é necessário que se definida um critério automático de parada.
Para auxiliar a determinação do critério de parada, admite-se que a obtenção da solução ótima para um dado problema nem sempre é essencial. Com efeito, admite-se que além de soluções ótimas, existam soluções suficientemente próximas destas que são satisfatórias do ponto de vista prático. Sendo assim, o objetivo principal passa de "encontrar soluções ótimas" para "encontrar soluções próximas da ótima dentro de um certo limite de tolerância".
Para determinar esse critério para *BLD*, procedeu-se um ensaio, no qual 20 problemas foram gerados aleatoriamente em um plano e resolvidos para diferentes valores do parâmetro *"a"*. Em cada problema a *BLD* foi aplicada para um número predeterminado de iterações, suficientemente grande, de modo a permitir que as soluções geradas se aproximassem da solução ótima. Os resultados observados serviram como base para a formulação de um modelo de regressão, no qual baseia-se um critério de determinação automática de *"a"*, e também, de parada do algoritmo.

O modelo gerado fornece uma estimativa prévia de quanto tempo (número de iterações) é necessário para o algoritmo atingir um patamar de proximidade da solução ótima (ou da melhor solução que a *BLD* poderia encontrar, frente ao número de iterações utilizado no ensaio).
É claro que qualquer que fosse o tempo utilizado na simulação, este poderia não ser o adequado para garantir a convergência da *BLD*. Até mesmo se utilizado tempo muito maior (infinito), o algoritmo poderia não convergir, pois não se trata de um algoritmo exato. Contudo, em testes previamente realizados para problemas de tamanhos similares aos dos considerados no ensaio, o algoritmo apresentou um elevado grau de precisão, convergindo com grande freqüência para valores muito próximos do ótimo. Portanto, se houver erros nas previsões geradas por esse modelo, espera-se que estejam dentro de um limite tolerável.
De posse dessas considerações, buscou-se uma estimativa empírica para "*a*" e um critério de parada em função do tamanho do problema proposto. Para cada problema gerado, o algoritmo BLD foi aplicado com um limite de iterações fixo conforme tabela III-2. A escolha desses limites para cada instância foi feita com base em simulações previamente realizadas sobre problemas com solução ótima conhecida.

Quantidade de problemas distintos e limite de iterações para cada um deles, segundo o tamanho do problema e para um mesmo nível de "*a*".

n	*Número de problemas*	*Limite de iterações*
200	5	200.000
400	5	400.000
600	5	600.000
800	5	800.000

Tabela IV-2

Número de problemas resolvidos, tempos totais e médios de execução utilizados, segundo o número de cidades do problema na simulação.

n	*Número de problemas*	*Tempo total de execução (horas)*	*Tempo médio por problema (horas)*
200	25	5,15	0,20
400	25	24,55	0,98
600	25	63,18	2,53
800	25	123,30	4,93
Total	100	216,18	-

Cada problema foi resolvido com o parâmetro "*a*" assumindo os valores 0,1; 0,2; 0,3; 0,4 e 0,5 e considerando a mesma rota inicial. Esses valores para "*a*" estão de acordo com os intervalos sugeridos na tabela III-1. O algoritmo foi implementado em Delphi 4.0, e executado em computador Pentium II – Celeron 333 MHz com 32 MB RAM, com índice de velocidade aferida pelo programa Norton Utilities 3.0, igual a 79[8]. Os tempos de simulação para os diferentes níveis de "*a*", tamanho do problema e grau de aproximação, encontram-se tabelas III-3 e III-4.
Foram geradas durante a simulação listas contendo: tamanho do problema (*n*); o tamanho da melhor solução até o momento, quando da finalização do processamento da *BLR*; a iteração em que a última melhora foi encontrada; o tempo gasto em que ocorreu a última melhora; e o valor do parâmetro utilizado. Para estes resultados, foi ajustado um modelo de regressão que associa o número de iterações, o valor do parâmetro e o tamanho do problema.

Tabela IV-3

[8] Um Pentium II – 300 MHz típico, alcança 140 pontos.

Tempos médios de execução (em segundos) observados, até que um nível de aproximação para a melhor solução conhecida, para os problemas testados, seja alcançado segundo o valor de "*a*" e o tamanho do problema.

		Parâmetro "a"				
%Aproximação	*n*	*0,1*	*0,2*	*0,3*	*0,4*	*0,5*
0,25	*200*	39,5	32,5	21,2	27,7	19,5
	400	185,8	237,1	233,7	157,5	205,3
	600	1.008,2	736,2	1.045,1	658,7	985,5
	800	2.414,4	1.682,1	1.838,5	1.453,5	1.402,4
0,50	*200*	15,7	13,6	13,5	16,9	15,3
	400	97,8	149,7	124,0	133,7	83,7
	600	408,8	335,0	278,1	298,1	359,9
	800	1.014,3	804,2	819,0	857,8	851,9
1,00	*200*	9,1	9,6	6,8	7,2	4,6
	400	76,4	44,9	55,9	51,9	46,6
	600	239,9	145,2	131,0	123,4	134,0
	800	493,4	330,1	279,6	333,2	302,4
2,00	*200*	4,4	3,8	3,0	3,5	2,9
	400	32,2	28,8	26,3	17,0	21,3
	600	89,9	52,7	55,2	41,8	52,7
	800	273,5	176,9	143,4	116,6	132,0
3,00	*200*	3,3	2,4	2,5	1,9	1,8
	400	20,3	17,8	15,1	11,5	14,6
	600	60,2	39,2	30,4	26,3	29,6
	800	119,6	89,9	77,6	70,8	65,6
4,00	*200*	2,7	1,8	1,7	1,4	1,2
	400	15,3	11,2	9,5	8,6	9,8
	600	40,5	27,5	20,6	17,6	20,2
	800	79,6	56,8	54,8	48,3	46,6
5,00	*200*	1,9	1,5	1,3	1,1	1,0
	400	12,4	8,0	7,8	7,0	7,2
	600	30,2	20,1	16,6	13,9	16,2
	800	63,0	43,0	37,0	36,0	33,4

Tabela IV-4

A fim de facilitar a explanação dos resultados obtidos, considere as seguintes variáveis:

- $s_{n,i}(a_j, k)$ representando a menor rota encontrada até a *k-ésima* iteração, onde a_j é o valor do parâmetro utilizado, *i* é o identificador do problema e *n* o tamanho da instância.
- $s^{\#}_{n,i}(a_j) = \min\{s_{n,i}(a_j, k) \mid 1 \le k \le máx.\ de\ iterações\ programadas\}$ é a menor rota obtida no *i-ésimo* problema utilizando o valor a_j como valor para o parâmetro *a* com $a_j \in \{0{,}1;\ 0{,}2;\ 0{,}3;\ 0{,}4;\ 0{,}5\}$.
- $s^{\#}_{n,i} = \min_{a_j} s^{\#}_{n,i}(a_j)$ é a menor rota obtida independente do valor do parâmetro, para o *i-ésimo* problema.
- $m_{n,i}(a_j, x) = \max\{k \mid s_{n,i}(a_j, k) / s^{\#}_{n,i} \le x; 1 \le k \le máx.\ de\ iterações\ programadas\}$ o número mínimo de iterações onde observou-se um distanciamento relativo a $s^{\#}_{n,i}$ menor ou igual a *x*, quando resolvido o *i-ésimo* problema com o parâmetro a_j.

Assim, cada $m_{n,i}(a_j, x)$, *i*=1,...,20 representa uma observação da variável *Número Mínimo de Iterações (m)*, que é a quantidade mínima de iterações necessárias para se alcançar um nível de distanciamento igual ou inferior a *x%* do valor da solução ótima (uma solução bastante próxima), fixado o valor a_j para o parâmetro *"a"* e *n* o tamanho do problema.
Consideradas as variações do parâmetro "*a*", de *n* e *x*, conforme já descrito, um total de *p*=700 observações foram feitas para a variável *Número Mínimo de Iterações.*
Com base nos valores obtidos na simulação, um modelo de regressão linear foi ajustado para a variável transformada *Y*=Ln(*Número Mínimo de Iterações*). A equação de previsão do modelo ajustado é dada por

$$\hat{Y} = 8{,}37914160669422 + 0{,}0079303162317396.\boldsymbol{n} - 4{,}89692161242129.\boldsymbol{a} - 1{,}28525509021886.\boldsymbol{x} - 0{,}000160445727116437.\boldsymbol{n}.\boldsymbol{x} - 0{,}53791360344216.\boldsymbol{a}.\boldsymbol{x} - 0{,}00000403456582016638.\boldsymbol{n}^2 + 6{,}46567682022229.\boldsymbol{a}^2 + 0{,}15512246462355.\boldsymbol{x}^2 \quad (1)$$

Para um modelo de regressão linear, o intervalo de confiança para os valores previsto é dado por

$$\hat{Y} \pm z_a . \left(s^2 \left(1 + \frac{1}{p} \right) + \sum_i \sum_j (x_{i0} - \bar{x}_i).(x_{j0} - \bar{x}_j).\mathrm{cov}(\hat{b}_i, \hat{b}_j) \right) \quad (2)$$

onde z_α é tal que $P(Z \leq z_\alpha) = 1-\alpha/2$ e a variável aleatória $Z \sim N(0,1)$, x_{i0} é o valor da *i-ésima* variável independente, $\bar{x}_i$ é a média aritmética dessa variável, p é o número de observações e $\hat{b}_i$ o coeficiente ajustado para a mesma variável, s^2 estimativa da variância do modelo e $\mathrm{cov}(\hat{b}_i, \hat{b}_j)$ a covariância entre os parâmetros estimados, Maddala [Mad98] ou Draper e Smith [Dra81]. A matriz de covariância e as médias das variáveis encontram-se em anexo.

A equação de previsão para a variável de interesse, *m*, é obtida através de $\hat{m} = e^{\hat{Y}}$.

Na figura III-1 é exibida a superfície de nível da função de previsão $\hat{m}$, fixado o valor do nível de aproximação em 1%, juntamente com as curvas de nível dessa superfície com níveis de aproximação fixados em 0,25, 0,5% e 1,0%.
Vê-se na figura, que para um nível de aproximação fixado em 1%, o crescimento do número mínimo de iterações é menor para valores de *"a"* pouco maiores que 0,4, isto é, um número menor de iterações é necessário para que o patamar de aproximação pré-fixado seja alcançado, quando são utilizados valores de *"a"* dessa magnitude. Através das curvas de nível, nota-se que, a redução do nível de erro aceitável também implica em uma redução no valor de *"a"*, caso se deseje manter o número de iterações sempre no menor patamar possível.

Estudando o comportamento da função de previsão $\hat{m}$, verifica-se que o valor do parâmetro "*a"* que minimiza as iterações, a^*, é dado por

$$a^* = \frac{4{,}89692161242129 + 0{,}53791360344216 \times x}{2 \times 6{,}46567682022229} \quad ; \; x \geq 0 \quad (3)$$

Observa-se que a equação de previsão (1) para o número mínimo de iterações é valida somente dentro dos limites em que as variáveis *x*, *n e "a"* foram estudadas. Portanto, previsões feitas para valores muito além dessa região, podem resultar em valores irreais, gerando uma calibração inadequada do algoritmo. De modo geral, quanto mais distante desses limites, pior. O mesmo alerta, vale para os valores gerados pela equação (3) deduzida a partir de (1).

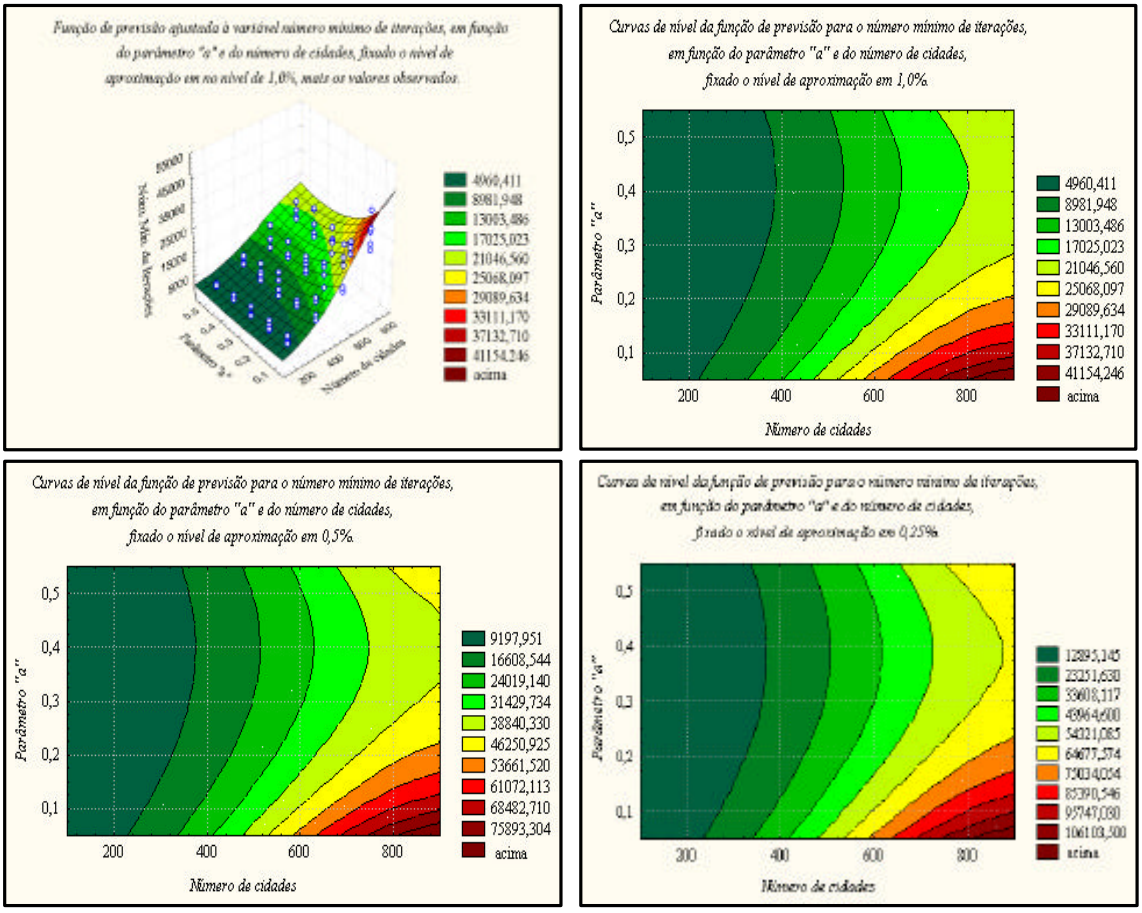


Figura III-1: Curvas e superfície de nível para a regressão ajustada à variável *número mínimo de iterações.*

3.7. Considerações finais

Define-se, então:

i) o ajuste do parâmetro *"a"* será feito conforme valor calculado pela equação (3); e

ii) o critério de parada será dado pela aplicação da função exponencial ao limite superior, do intervalo gerado pela equação (2), utilizando α=0,05.

Capítulo V

2. Resolução do PCV de Grandes Dimensões Através de Particionamento

2.1. Considerações iniciais

Como já mencionado, em muitas situações práticas não é necessário determinar a solução ótima de um PCV. Usualmente, é mais importante encontrar uma solução com uma relação favorável entre qualidade e o tempo necessário para sua obtenção, isto é, deve-se buscar uma solução viável e satisfatória, em algum sentido, no menor tempo possível.
Considerando algoritmos aproximados, os maiores problemas que se tem notícia, tratados como um único bloco, tinham 1.000.000 de cidades. Utilizando uma abordagem de particionamento, similar a que será descrita mais adiante, o número se eleva para 10.907.064 cidades. Neste último caso, o problema original foi divido em mais de 1.000 subproblemas, e a solução foi obtida com a utilização de uma rede de computadores trabalhando por duas semanas. Desse esforço, o resultado obtido mostrou-se distante do limite inferior de Held/Karp por apenas 4,9 %.

2.2. Heurísticas de agrupamento (cluster)

A idéia central de heurísticas de agrupamento está em particionar um problema intratável, em vários subproblemas menores e mais fáceis de resolver. Uma vez resolvido cada um desses subproblemas, um processo é utilizado para unir essas soluções elementares e gerar uma solução para o problema principal.

2.2.1 Vantagens e desvantagens

De modo geral, as principais vantagens oferecidas por procedimentos desse tipo são: o aumento na velocidade de resolução e a diminuição da exigência por recursos do sistema utilizado no processamento. Por exemplo, para se acelerar o processo de obtenção da solução no caso do PCV simétrico euclidiano, e forçosamente no caso de PCV com matriz de distâncias aleatórias, as distâncias entre todos os pontos são previamente calculadas ou fornecidas e armazenadas em uma estrutura de dados adequada. No caso do PCV simétrico, essas distâncias podem ser armazenadas em uma matriz triangular, que consome espaço na ordem de $n´(n-1)/2$ vezes, a quantidade de memória necessária para armazenar o valor da distância entre um par de cidades. Utilizado esse tipo de implementação, o consumo de memória é muito alto, mesmo para um número moderado de vértices (5.000, por exemplo). Outra questão é que, algumas heurísticas como busca tabu ou *BLD*, por exemplo, utilizam outras estruturas específicas dos métodos, como a lista tabu e matriz de penalidades respectivamente, que também concorrem por espaço na memória. Com o particionamento, esse problema é sensivelmente atenuado uma vez que com uma a implementação adequada, pode-se limitar a quantidade de memória necessária ao total exigido para resolver o maior problema do particionamento.
Outra vantagem que surge com o particionamento é a facilidade de utilização de processamento paralelo. Havendo necessidade e hardware disponível, está aberto o caminho para o aumento na velocidade de obtenção de soluções.
No quesito tempo, o particionamento apresenta comportamento tal que o tempo de processamento seqüencial cresce limitado superiormente por $T.n_k$, onde T é o tempo necessário para resolver o agrupamento mais difícil e n_k o número total de agrupamentos.
Como desvantagem principal, algoritmos que utilizam estratégias de particionamento apresentam uma perda na qualidade das soluções oferecidas.
Não obstante, parece ser a única abordagem possível na presença de limites de tempo muito curtos, de limitação de equipamentos ou ainda quando se tratar de problema demasiadamente grande.

2.2.2 Duas formas de particionamento

- *Particionamento geométrico*: Dado um conjunto de *n* vértices, determina-se o menor retângulo capaz de envolver todos os pontos. Em seguida, subdivide-se este em *k* retângulos iguais. Para um conjunto de vértices, cujas coordenadas são dadas por (x_i, y_j) isto é obtido fazendo

  $x_{min}=\min\{x_i\}; x_{max}:=\max\{x_i\}; y_{min}=\min\{y_i\}$ e $y_{max}=\max\{y_i\}$

  define-se $l \times h$ retângulos $R_{i,j}$ de largura $\frac{1}{l}.(x_{\max} - x_{\min})$ e altura $\frac{1}{h}.(y_{\max} - y_{\min})$ com o canto esquerdo inferior no ponto $(x_{\min} + \frac{i}{l}(x_{\max} - x_{\min}), y_{\min} + \frac{j}{h}(y_{\max} - y_{\min}))$.
- *Particionamento de Karp:* Em 1977, *Karp* [Kar77] propõe uma heurística que gera um conjunto de agrupamentos de modo recursivo, a partir de um retângulo inicial onde todos os vértices estão inscritos, dividindo-o pelo vértice mediano e de modo paralelo ao lado de menor comprimento. Os retângulos gerados vão sendo divididos sucessivamente, utilizando o mesmo mecanismo até o momento em que um número preestabelecido de divisões se faça. Terminado o processo de divisão a rota final é estabelecida através da aplicação de duas operações elementares.

2.3. Algoritmo de particionamento proposto

O esquema que será proposto é baseado no particionamento geométrico. O procedimento como um todo pode ser dividido em três fases:
Fase 1: Geração de grupos disjuntos de cidades.
Fase 2 : Determinação de boas rotas dentro de cada agrupamento.
Fase 3 : Unificação das rotas.
A seguir, a descrição de cada uma dessas fases:

- ***Fase 1:*** *Geração de grupos disjuntos de cidades.*

Para um problema com *N* cidades e com número máximo e mínimo, k_{max} e k_{min} respectivamente, execute o procedimento ***Particiona***, descrito a seguir.

Procedimento *Particiona*;
Início

Armazene as cidades em C_0 {coordenadas (x, y)}.

Leia k_{max} e k_{min}*: =#mín. cid. por agrupamento.*

Crie uma lista de conjuntos *C* e adicione C_0 a lista.
Enquanto $\exists\ C_i \in C, i=1,..,card(C); card(Ci)>k_{max}$
  Faça *Subdivide*(C_i,C, k_{max}).
FimEnquanto
Calcule o centro de gravidade para todos os conjuntos em *C.*
i:=1
Repita
  Se $card(C_i)< Max\{3, k_{min}\}$
  Então
    Transfira cada elemento de C_i, para o agrupamento *Cj* mais próximo[9].
    Remova C_i da lista *C*.
  Senão
    i: =i+1

[9] Define-se o agrupamento mais próximo de um elemento, como sendo aquele que apresenta a menor distância para o elemento tomada a partir de seu centro de gravidade, isto é são calculadas as distâncias entre o elemento que se quer realocar e centro de gravidade de todos os demais agrupamentos. Aquele para o qual essa distância é mínima, é o mais próximo.

FimSe
Até $i>card(C)$
Fim

Procedimento *Subdivide(C_i,C, k_{max})*
Início
Para C_i, calcule $dBasico=sqrt((x_{max}- x_{min})*(y_{max}-y_{min}))* k_{max}/Card(C_i)$.
Se $x_{max}= x_{min}$ ou $y_{max}=y_{min}$
Então
Se $x_{max}=x_{min}$ Então Faça $Ny:=2$ e $Nx:=1$
Se $y_{max}=y_{min}$ Então Faça $Ny:=1$ e $Nx:=2$.
Senão
$Nx:= Int(((x_{max}- x_{min}) / dBasico)+1)$
$Ny := Int(((y_{max}-y_{min}) / dBasico)+1)$.
FimSe
Se $(Nx=1)$ e $(Ny=1)$
Então Se Random>0.5
Então
$Nx: =2$
Senão
$Ny: =2$.
FimSe
Faça $dx: =(x_{max}- x_{min})/Nx$ e $dy:=(y_{max}-y_{min})/Ny$.
Divida C_i em Nx ´ Ny retângulos com largura dx e comprimento dy com canto inferior esquerdo no ponto $(x_{min}+(x_{max}-x_{min})*i/dx, y_{min}+(y_{max}-y_{min})*j/dy)$ associando todos os pontos compreendidos pelos limites desse retângulo ao agrupamento $C_{i, j}$.
Remova C_i de C e adicione todos os $C_{i,j}$ não vazios a C.
Fim.
Em linhas gerais, os procedimentos acima dividem as cidades em blocos limitados por áreas retangulares, tentando respeitar o valor de k_{max}. Se após uma primeira divisão restarem alguns agrupamentos com mais de k_{max} elementos, esses são recursivamente divididos até que esse limite de cidades seja alcançado. Ao fim desse processo, são acrescentados na lista de agrupamentos, *C*, somente aqueles agrupamentos que sejam não vazios.
Inicia-se em seguida, um processo de eliminação de agrupamentos com número inferior ao $máx\{3, k_{min}\}$. Esse processo tem dois objetivos: primeiro, tornar mais homogêneo o número de elementos distribuídos entre os diversos agrupamentos; e segundo, garantir um número mínimo de elementos em cada agrupamento que viabilize a construção de uma rota.
Isso é feito, identificando-se em *C*, o primeiro agrupamento (conforme a posição que ocupa na lista *C*), com menos que $máx\{3, k_{min}\}$ elementos. Então, inicia-se a relocação de seus elementos para os agrupamentos mais próximos a cada um deles, e que contenham pelo menos 3 elementos. Esvaziado o agrupamento, ele é removido de *C*. Esse processo se repete até que em C restem apenas agrupamentos com pelo menos $máx\{3, k_{min}\}$.
Vale lembrar, os centros de gravidade dos grupos não são recalculados com a entrada desses novos elementos.
***Fase 2:** Determinação de rotas dentro de cada agrupamento.*
Nesta fase, procede-se para cada um dos agrupamentos a construção de uma rota para as cidades do mesmo, utilizando o algoritmo para o PCV descrito no capítulo III, junto com o critério de ajuste de do parâmetro *"a"* e regra de parada descrita na seção 3.6.
Adicionalmente, utilizam-se listas de vizinhos, como descrito na seção 2.4.2.3.1 do capítulo II, com o objetivo de acelerar o processo de busca local *2-opt*, utilizado no algoritmo *BLD*.
❖ ***Fase 3**: Unificação das rotas.*
Finalmente, a geração de uma solução para o problema inicial é feita aplicando-se o procedimento GeraRotaUnificada(n_v) descrito abaixo:
Procedimento *GeraRotaUnificada(n_v);*
Início

Para cada agrupamento C_i em C, construa uma lista que indique os n_v agrupamentos mais próximos[10] de C_i. {Geração dos agrupamentos vizinhos de C_i}
Para cada par C_i e C_j vizinhos, determine os pontos $c_i \hat{I} C_i$ e $c_j \in C_j$ mais próximos.
Determine as ligações mais econômicas entre as rotas de cada par de vizinhos.
Construa um grafo G, onde cada nó representa um agrupamento, com um arco para cada par de agrupamentos vizinhos. Associe a cada arco o valor da ligação mais econômica entre os pares que geraram o arco.
Determine uma árvore mínima sobre G.
Gere a rota final realizando as ligações entre os agrupamentos de acordo com os arcos determinados na árvore mínima calculada.

Fim.

Sejam c_i e c_j os pontos mais próximos entre os agrupamentos C_i e C_j. Sejam também, R_i e R_j, as rotas para esses mesmos agrupamentos respectivamente, e determinadas na fase 2. A conexão mais econômica entre cada par de agrupamentos vizinhos é determinada pela conexão de menor custo, dentre todas as possíveis conexões que levam em conta a remoção do arco ligando c_i e seu sucessor (antecessor) na rota R_i, e também a remoção do arco ligando c_j e seu sucessor (antecessor) na rota R_j. O custo da conexão é determinado pela diferença de custo entre os arcos que estão sendo criados e os que estão sendo removidos. A figura IV-1 representa uma dessas possibilidades.
Na implementação realizada se utilizou o algoritmo de Prim, descrito em Christofides [Chr75], na fase de construção da árvore mínima para o grafo gerado, G.

Em problemas onde os agrupamentos cidades formados são muito distantes uns dos outros, é possível que o grafo G seja desconexo. Neste caso, não é possível construir uma árvore mínima. Este contratempo pode ser resolvido facilmente aumentando o

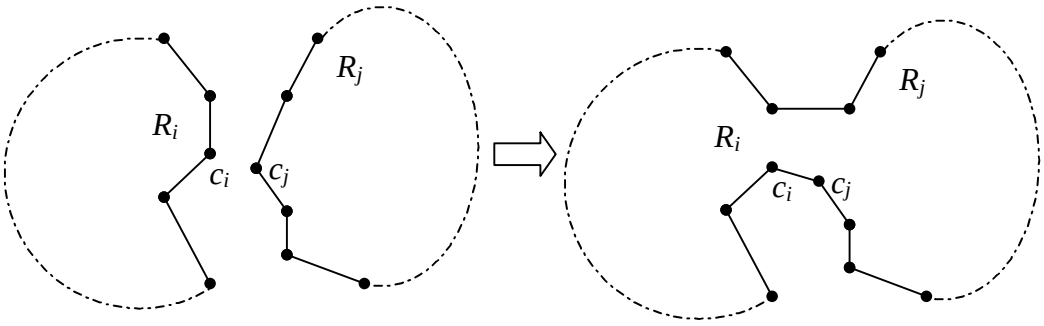


Figura V-1 : Exemplo de conexão entre duas rotas

valor de n_v.

2.4. Testes numéricos

A fim de avaliar a qualidade das soluções geradas pelo algoritmo, um conjunto de problemas disponíveis na biblioteca TSPLIB (biblioteca de problemas, amplamente utilizada para comparar o desempenho de diferentes técnicas, destinadas à resolução do PCV, Reinelt [Rei91], disponível também via internet em ftp://softlib.rice.edu) foram selecionados para testes.
Os problemas escolhidos apresentam tamanhos variando entre 1.000 e 14.051 cidades. São problemas reais e relacionados à perfuração de placas de circuitos e a minimização de rotas entre cidades reais entre outros.
O algoritmo descrito na seção anterior foi implementado em Delphi 4.0. Todos os testes foram realizados em um computador com processador AMD-K6/2 de 350 MHz e 64Mb de memória principal, rodando

[10] A proximidade entre um par de agrupamentos é dada pela distância entre seus centros de gravidade.

com o sistema operacional Windows 98. A velocidade desse equipamento aferida pelo software Norton Utilities 3.0, alcança índice médio igual a 100[11].

2.5. Apresentação dos problemas.

Os problemas selecionados foram classificados de acordo com a distribuição dos pontos no plano, em três categorias: os uniformes, que apresentam uma distribuição aleatória e de densidade uniforme no plano; os regulares, que apresentam formas geométricas bem definidas; e os com clusters, que apresentam uma configuração onde se percebem conjuntos isolados de pontos. Os problemas, as classificações e as rotas ótimas (ou limite inferior) estão dispostos na tabela IV-1. A representação geométrica desses problemas é exibida na figura IV-2.
Para cada problema foram realizadas diversas execuções com diferentes valores para os parâmetros k_{max}, k_{min}, *número de iterações*, *número de cidades vizinhas*, *número de agrupamentos vizinhos* e quando utilizado o ajuste sugerido no capitulo III, diferentes valores para o *nível de aproximação*.

Problemas selecionados da TSPLIB

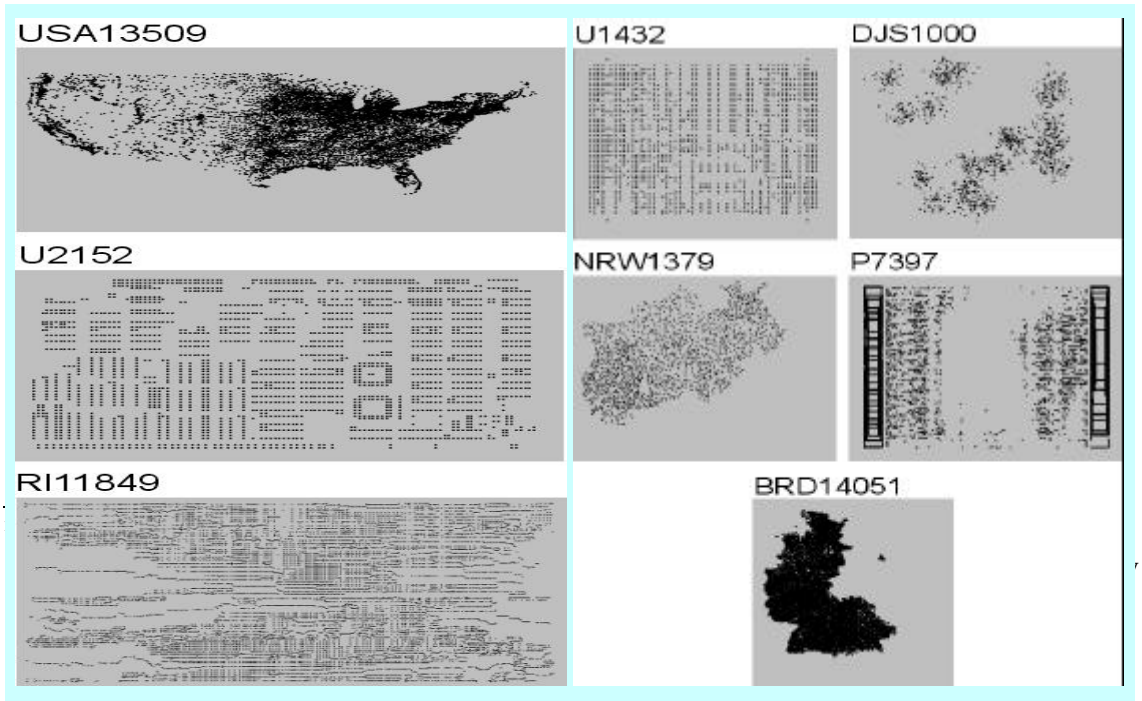
Problema	*#Cidades*	*Forma*	*Solução ótima*
DJS1000	1.000	Com clusters	18.659.688
NRW1379	1.379	Uniforme	56.638
U1432	1.432	Regular	152.970
U2152	2.152	Regular	64.253
PLA7397	7.397	Com clusters	23.260.728
RL11849	11.849	Uniforme	920.847*
USA13509	13.509	Uniforme	19.947.008*
BRD14051	14.051	Uniforme	468.942*

* Limite inferior

Tabela V-1

Compararam-se também os tempos e as soluções obtidas, com as de um outro algoritmo também baseado em estratégia de particionamento. Na questão qualidade da solução, a comparação é clara. Entretanto, no quesito tempo, esta deve ser feita com maior cautela, pois os algoritmos são implementados de maneira diversa (sujeito às habilidades do programador), existem diferenças de desempenho dos compiladores utilizados e, principalmente, diferenças entre as capacidades de processamento do hardware utilizado. Em todo caso fica como referência.
Para avaliar a precisão dos resultados gerados pelo algoritmo, são feitas comparações, calculando-se o excesso para a solução ótima definido por

Problemas selecionados da TSPLIB



$$excesso = \frac{Solução\ gerada - Solução\ ótima}{Solução\ ótima} \times 100,$$

se esta é conhecida. Caso contrário, utiliza-se o limite inferior fornecido para o problema na TSPLIB.

Quanto ao quesito tempo, a comparação é um pouco mais difícil de ser feita. No trabalho de Bachem *et al* [Bac94], que adota também estratégia de particionamento, são fornecidas informações necessárias à comparação com os problemas BRD14051 e RL11849. Por essa razão, apenas para esses problemas, isso será feito. Os demais serão apenas comparados com a TSBLIB.

O algoritmo proposto por Bachem *et al* [Bac94] utiliza como estratégia de geração de grupos o particionamento de Karp [Kar77], e como heurística de melhoria de rotas uma implementação paralela do algoritmo LK. O equipamento utilizado foi um Parsytec Gcel 3/1024 com memória distribuída e 1024 processadores T805, com 4 MB de RAM cada um. No processo de divisão utilizado, criaram agrupamentos com 750 cidades para os problemas BRD14051 e RL11849. Os resultados que obtiveram estão dispostos na tabela IV-2

Resultados de Bachem *et al*

Problema	*Agrupamentos*	*Tempo seqüencial*	*# processadores utilizados*	*Solução*	*Excesso*
RL11849	32	2.413,05	16	972.338	5,55
BRD14051	32	2.350,87	16	486.435	3,73

Tabela V-2

O algoritmo foi testado em três configurações diferentes:

i) todos os problemas foram divididos em agrupamentos com no máximo 1.400 cidades e o algoritmo aplicado conforme descrito na seção 4.3 deste capítulo,. Resultados na tabela IV-3;

ii) problemas com menos de 3.000 cidades foram resolvidos como um único agrupamento com número de iterações fixado em 70.000 e com parâmetro a=0,389. Resultados na tabela IV-4; e

iii) problemas com mais de 7.000 cidades foram divididos em agrupamentos contendo mais de 1000 cidades cada e com número iterações fixado em 70.000 e com parâmetro a= 0,389. Resultados na tabela IV-5.

Como se nota, para a configuração (i), a precisão máxima obtida em todos os problemas testados o excesso ficou compreendido no intervalo 1,23 a 4,53.

Melhores resultados obtidos utilizando o procedimento *Particionamento+BLD+BLR+2-opt* com ajuste automático dos parâmetros, para agrupamentos com número limite de cidades igual a 1.400.

Características	*DJS1000*	*NRW1379*	*U1432*	*U2152*
#Agrupamentos	3	2	4	3
Menor/maior agrupamento	245/398	588/791	307/393	658/765
#Agrupamentos vizinhos	5	5	5	5
#Cidades vizinhas	20	20	20	20
Aproximação	0,25	0,25	0,25	0,25
Melhor solução	18.958.252	57.471	154.857	65.506
Tempo (s)	33,3	162,2	53,0	260,0
Excesso (%)	1,60	1,47	1,23	1,95
Características	*PLA7397*	*RL11849*	*USA13509*	*BRD14051*
Menor/maior agrupamento	1.096/1.361	629/1.285	345/822	588/1106
#Agrupamentos vizinhos	10	5	5	5
#Cidades vizinhas	20	20	20	20
Aproximação	0,25	0,25	0,25	0,25
Melhor solução	23.698.756	962.561	20.766.205	480.888
Tempo (s)	1.028,3	1.821,20	1.040,9	2.090,7
Excesso (%)	1,88	4,53	4,11	2,55

Tabela V-3

Comparando os resultados obtidos para os problemas BRD14051 e RL11849, por Bachem *et al* [Bac94] (tabela IV-2) e a resolução pela forma descrita em (i), verifica-se uma redução no excesso a favor do procedimento proposto de 1,18% e 1,02% para cada um dos problemas, respectivamente. Quanto ao tempo, verifica-se uma redução de 11% e de 24,5% na mesma ordem.

Melhores resultados obtidos utilizando o procedimento *BLD+BLR+2-opt* com limite fixo de iterações igual a 70.000.

Características	*DJS1000*	*NRW1379*	*U1432*	*U2152*
#Cidades vizinhas	20	20	20	20
Melhor solução	18.958.252	57.127	153.867	64.761
Tempo (s)	184,1	199,7	138,7	326,3
Excesso (%)	1,34	0,86	0,59	0,79
Características	*PLA7397*	*RL11849*	*USA13509*	*BRD14051*
#Agrupamentos	4	7	8	8
Menor/maior agrupamento	1.546/1.988	1.361/2.104	1.035/2.260	1.363/2.397
#Agrupamentos vizinhos	5	5	5	5
#Cidades vizinhas	20	20	20	20
Solução	23.667.818	954.493	20.515.036	482.020
Tempo	1.169,3	2.036,0	2.042,7	2.273,3
Excesso	1,75	3,65	1,72	2,57

Tabela V-4

Melhores resultados obtidos utilizando o procedimento *Particionamento+BLD+BLR+2-opt* com limite fixo de iterações por partição igual a 70.000 em problemas grandes.

Tabela V-5

Comparando as abordagens (i) e (ii), observa-se uma redução média de 0,67 no excesso e um aumento percentual médio de 33% no tempo de processamento pela utilização da abordagem (ii).
Para problemas com mais de 7.000 cidades, comparando–se as diferenças entre as abordagens (i) e (iii), observa-se uma redução média no excesso de 0,84 pontos e um aumento no tempo médio de resolução igual 34,1% com a utilização da abordagem (iii).
Comparando os resultados obtidos por Bachem *et al* [Bac94] (tabela IV-2) e a resolução desses pela forma descrita em (iii), verifica-se uma redução média 1,53 no excesso e de 9,54% no tempo médio de processamento, em favor do procedimento (iii).

2.6. Considerações finais

Observa-se que o procedimento de particionamento proposto foi capaz de obter soluções com excesso máximo 3,65 para todos os problemas estudados, em um tempo inferior a 40 minutos.

Capítulo VI

2. Conclusões e Recomendações

2.1. Considerações finais

Neste trabalho foram apresentadas as técnicas exatas e heurísticas mais utilizadas na resolução do Problema do Caixeiro Viajante na atualidade. Particularmente, o enfoque se concentrou nas chamadas técnicas heurísticas, por apresentarem características que favorecem a sua utilização em ambiente com poucos recursos computacionais ou com limitação de tempo para a resolução do problema, um dos objetivos desse trabalho.
Com o objetivo de acelerar o processo de obtenção de soluções e de reduzir a exigência do poder de processamento do hardware envolvido na resolução do PCV, foi proposto, implementado e avaliado um algoritmo baseado em estratégia de particionamento, utilizando em conjunto a metaheurística *Busca Local Dirigida*.
Tal metaheurística mostrou-se adequada na resolução do PCV, apresentando como características de destaque: simplicidade de programação e geração de soluções de qualidade com rapidez.
Testes realizados com o algoritmo proposto, sobre problemas da TSPLIB de Reinelt, deram indícios de que o algoritmo desenvolvido é competitivo, principalmente em situações onde os problemas não apresentam uma formação nítida de clusters.
De modo geral, o algoritmo foi capaz de gerar "bons" resultados em "curto" espaço de tempo. Nos testes realizados, apresentou apreciável grau de precisão, gerando excessos na ordem de 2 a 3% para problemas de grande porte (mais de 2000 nós) e menos de 1% para problemas de médio porte (200 a 2000 nós). Para problemas de pequeno porte (menos de 200 nós), os erros observados são praticamente desprezíveis. Porém, conforme esperado, esse mesmo procedimento não foi capaz de igualar a qualidade das soluções geradas quando se ataca o problema como um único bloco, isto é, sem a utilização de estratégias de particionamento.

2.2. Recomendações

Aos interessados em desenvolver trabalhos futuros com base no algoritmo aqui proposto, recomenda-se:

- um reajuste da função de ajuste automático de parâmetros, descrita no capítulo III, de forma a ampliar seu intervalo de utilização (repetir a simulação para problemas com maior número de cidades), e eventualmente, reformular o modelo, de modo a considerar o tamanho da lista de vizinhos utilizada para as cidades.
- observou-se que a qualidade das soluções tende a crescer à medida em que se aumenta o tamanho das partições. Assim, a utilização de estruturas dados mais eficientes e econômicas para armazenamento de rotas e distâncias, como as tratadas por Fredman *et al* [Fre95], podem ser consideradas. Tais estruturas permitem a manipulação de problemas de dimensões maiores, com maior eficiência que a oferecida por estruturas de vetores (como as utilizadas na implementação do procedimento proposto nesse trabalho). Portanto, a utilização dessas estruturas deverá produzir ganho no desempenho do algoritmo.
- criar um mecanismo que permita a interação do usuário na fase formação das partições, poderá resultar em ganho na qualidade das soluções produzidas, principalmente em problemas com clusters bem definidos.

Bibliografia

[Aar97] AARTS, E. H. L., KORST, J. H. M. e van LAARHOVEN, J. M. (1997). "Simulated Annealing". Aarts, E. e Lenstra, J. K. (eds). *Local Search in Combinatorial Optimization*, Wiley, New York, 91-120.

[App95] APPLEGATE, D., BIXBY, R. E., CHVÁTAL, V., COOK, W. (1995) "Finding Cuts in the TSP: a Preliminary Report", Report 95-05, DIMACS, Rutgers University, New Brunswick, NJ.

[Bac94] BACHEM, A., STECKEMETZ, B. e WOTTAWA, M. (1994). "An Efficient Parallel Cluster-heuristic for Large Travelling Salesman Problems", *Technical Report* 94-150, Universität zu Köln.

[Bal85] BALAS, E. e TOTH, P., (1985) "Branch and Bound Methods", (eds) LAWLER, E. L., LENSTRA, J. K., RINNOOY, A. H. G. e SHMOYS, D. B. *The Travelling Salesman Problem – A Guided Tour in Combinatorial Optimization*, Wiley, New York, 361-401.

[Bar88] BARTHOLDI II, J. J., PLATZMAN, L. K. (1988) "Heuristic Based on Spacefilling Curves of for Combinatorial Problems in Euclidean Space". *Management Science* **34** (3), 291-305.

[Ber73] BERGE, C. (1973). "*Graphs and Hypergraphs"*. North-Holland, Amsterdam.

[Bla89] BLAND, R. G. e SHALLCROSS, D. F. (1989). "Large Travelling Salesman Problems Arising Form Experiments in X-Ray Crystalography. A Preliminary Report on Computation". *Operations Researt Letters* **8**, 125-128.

[Bod83] BODIN, L. D., GOLDEN, B. L., ASSAD, A. e BALL, M. (1983). "Routing and Scheduling of Vehicles and Crews". *Computeres and Operations Research*, **10**, 69-211.

[Bon84] BONOMI, E. e LUTTON, J. (1984). "The N-city Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm". *SIAM Review* **26**, 551-568.

[Cha96] CHATTERJEE, S.; CARRERA, C.; LYNCH, L. A (1996). "Genetic Algorithms and Travelling Salesman Problems". *European Journal of Orperations Research* **93**, 490-510.

[Chr69] CHRISTOFIDES, N e ELION, S. (1969). "An Algorithm for the Vehicle-Dispathching Problem". *Operarional Research Quaterly* **20**, 309-318.

[Chr75] CHRISTOFIDES, N. (1975). *"Graph Theory: An Algorithm Approach"*. Academic Press. Inc.

[Cla64] CLARKE, G. e WRIGHT, J. (1964). "Scheduling of Vehicles from a Central Depot to a Number of Delivery Points". *Operations Research* **12**, 568-581.

[Cro58] CROES, G. (1958). "A Method for Solving Travelling Salesman Problems", *Operations Research* **6**, 791-8112.

[Cro80] CROWDER, H. e PADBERG, M. W. (1980) "Solving Large-scale Symmetric Travelling Salesman Problems to Optimality", *Management Science* **26**, 495-509.

[Dan54] DANTZIG, G. B., FULKERSON, D. R. e JOHNSON, S. M. (1954), Solution of a Large-scale Travelling Salesman Problem, *Operations Research* **2**, 393-410.

[Dra81] DRAPER, N. R. e SMITH, H. (1981). *"Applied Regression Analysis"* – 2nd ed., Wiley, New York.

[Eis91]EISELT, H. A. e LAPORTE, G. (1991). "A Combinatorial Optmization Problem Arising in Dartboard Design", *Journal of the Operational Research Society* **42**, 113-118.

[Fre95] FREDMAN , M. L., JOHNSON, D. S., McGEOCH, L. A. e OSTHEIMER, G. (1995) "Data Structures for Travelling Salesmen". *J. Algorithms* **18**, 432-479

[Frei96a] FREISLEBEN, B. e MERZ, P. (1996) "New Genetic Local Search Operators for the Travelling Salesman Problems", *in Proceddings of 4th Conference or Paralallel Problem Solving from Nature* – PPSN IV

(Voigt, H. M., Ebeling, W. Rechenberg, I. e Schweal, P.), Springer, 890-900

[Frei96b] FREISLEBEN, B. e MERZ, P. (1996) "New Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Travelling Salesman Problems", *in Proceddings of the 1996 IEEE International Conference on Evolucionary Computation* (Nagoya, Japan), 890-900

[Gal97] GALVÃO, R. D., BARROS NETO, J. F., FERREIRA FILHO, V. J. M. e HENRIQUES, H. B. S. (1997). "Roteamento de Veículos com Base em Sistemas de Informação Geográfica". *Gestão e Produção* **4**, 159-173.

[Gar79] GAREY, M. R. e JOHNSON, D. S. (1979). "*Computers and Intractability. A Guide to the Theory of NP-Completeness"*, Freemann, New York.

[Garf77] GARFINKEL, R. S. (1977). "Minimizing Wallpaper Waste, Part I: A Class of Travelling Salesman Problems", *Operations Research* **25**, 741-751.

[Gen92] GENDREAU, M., HERTZ, A. e LAPORTE, G. (1992). "New Insertion and Postoptimization Procedures for the Travelling Salesman Problem". *Operations Research* **40**, 1086-1094.

[Gen94] GENDREAU, M., HERTZ, A. e LAPORTE, G. (1994). "A Tabu Search Heuristic for the Vehicle Routing Problem". *Management Science* **40**, 1276-1290.

[Gil74]GILLETT, B. E. e MILLER, L. R. (1974). "A Heuristic Algorithm for the Vehicle-Dispatch Problem". *Operations Research* **22**,340-349.

[Gol80] GOLDEN, B., BODIN, L., DOYLE, T. e STEWART JR, W. (1980). "Aproximate Travelling Salesman Algorithms". *Operations Research* **28**(3), 694-711.

[Gol85] GOLDEN, B. e STEWART, W. R. (1985). "Empirical Analisys of Heuristics". (eds) LAWLER, E. L., LENSTRA, J. K., RINNOOY, A. H. G. e SHMOYS, D. B. *The Traveling Salesman Problem – A Guided Tour in Combinatorial Optimization*, Wiley, New York, 203-249.

[Gro80] GRÖTSCHEL, M. (1980) "On the Symmetric Travelling Salesman Problem: Solution of a 120-city Problem", *Mathematical Programming Studies* **12**, 67-77.

[Gro88] GRÖTSCHEL, M., LOVÁSZ, L. e SCHRIJVER, A. (1988) "*Geometric Algorithms and Combinatorial Optimization*", Springer-Verlag.

[Her97] HERTZ, A.; TAILARD, E. e WERRA, D. (1997). "Tabu Search". Aarts, E. and Lenstra, J. K. (eds). *Local Search in Combinatorial Optimization*, Wiley, New York, 121-136.

[Hjo95] HJORRING, C. A. (1995), "The Vehicle Routing Problem and Local Search Meta-Heuristics", Ph. D. Thesis, Departament of Engeneering Science, The University Auckland, NZ.

[Hof93] HOFMANN, R.(1993) Ph.D Thesis. "Examinations on the Algebra of Genetic Algorithms", Technische Universität Müchen, Institut für Informatik.

[Hol75] HOLLAND, H. J. (1975). "Adaption in Natural and Artificial Systems", MIT Press, Cambridge, MA.

[Joh97] JOHNSON, D. S. e McGEOCH, L. A. (1997). "The Travelling Salesman Problem: a Case Study". Aarts, E. e Lenstra, J. K. (eds). *Local Search in Combinatorial Optimization*, Wiley, New York, 215-310.

[Jun93] JÜNGER, M. , REINELT, G. e THIENEL, S. (1993). "Provably Good Solutions for Travelling Salesman Problem". *Technical Report*, Universtität zu Köln.

[Kar71] HELD, M. e KARP, R. M. (1971). "The Travelling-Salesman Problem and the Minimun Spanning Trees: Part II", *Mathematical Programming* **6**, 6-25.

[Kar77] KARP, R. M. (1977). "Probabilistic Analisys of Partitioning Algorithms for the Travelling-Salesman Problem in the Plane", *Mathematics of Operations Research*, **2**, 209-224.

[Kir83] KIRKPATRICK, S., GELATT Jr., C. D. e VECCHI, M. P. (1983). "Optimization by Simulated Annealing". *Science* **220**, 671-680.

[Kno94] KNOX, J. (1994). “Tabu Search Performance on the Symmetric Travelling Salesman Problem”. *Computers Ops. Res.* **21**, 867-876.

[Lap92a] LAPORTE, G. (1992). “The Traveling Salesman Problem: An overview of Exact and Approximate Algorithms”. *European Journal of Operations Research* **59**, 231-247.

[Lap92b] LAPORTE, G. (1992). “The Vehicle Routing Problem: An Overview of Exact and Aproximate Algorithms”. *European Journal of Operational Research* **59**, 345-358.

[Lin65] LIN, S. (1965). “Computer Solutions for the Travelling Salesman Problem”. *Bell Syst. Computing Journal* **44**, 2245-2269.

[Lin73] LIN, S. e KERNIGHAN, B. W. (1973). “An Effective Heuristic Algorithm for the Travelling Salesman Problem”. *Operations Research* **21**, 498-516.

[Mad98] MADDALA, G. S. (1998). “*Introduction of Econometrics”* – 2nd ed., Prentice-Hall, New Jersey.

[Met53] METROPOLIS, N., ROSENBLUTH, M., TELLER, A. e TELLER, E. (1953), “Equation of State Calculations by Fast Computing Machines”, *J. Chem. Phys.* **21**, 1087.

[Osm96] OSMAN, I. H. e LAPORTE, G. (1996), “Metaheuristics: A bibliography” *Annals of Operations Research* **63**, 513-623.

[Pad91] PADBERG, M.W. e RINALDI, G. (1991) “A Branch and Cut Algorithm for the Resolution of Large-scale Symmetric Travelling Salesman Problems, *SIAM Review* **33**, 60-100.

[Pae88] PAESSENS, H. (1988). “The Savings Algorithm for the Vehicle Routing Problem”. *European Journal of Operations Research* **34**, 336-344.

[Pot96] POTVIN, J. (1996). “Genetic Algorithms for the Travelling Salesman Problem”. *Annals of Operations Research*, **63**, 339-370.

[Ree96] REEVES, C. R. (1996) “Modern Heuristic Technics” . RAYWARD-SMITH, OSMAN, I. H., REEVES, C. R. e SMITH, G. D. (eds). *Modern Heuristic Search Methods*, John Wiley & Sons, 1-25.

[Rei91] REINELT, G. (1991). “TSPLIB: a Travelling Salesman Problem Library”. *ORSA Jornal on Computing* **4**, 86-384.

[Ros77] ROSENKRANTZ, R. S. e LEWIS, P. (1977). “An Analisys of Several Heuristics for the Travelling Salesman Problem”. *SIAM Journal Computing* **6**, 563-581.

[Sch98] SCHMITT, L. J., AMINI, M. M. (1998). “Performance Characteristics of Alternative Genetic Algorithmic Approcches to the Travelling Salesman Problem Using Path Representation: An Empirical Study”. *European Journal of Operations Research* **108**, 551-570.

[Sys91] SYSWERDA, G. (1991). “Reproduction in Generational and Steady State Genetic Algorithm” In: Ratlines, G. (ed.) *Foundations of Genetic Algorithms*, Morgan Kaufmann, Los Altos, CA, 94-101.

[Vou97] VOUDOURIS, C. (1997). “Guided Local Search for Combinatorial Optimizations Problems”, Ph. D Thesis, Departament of Computer Science, University of Essex, Colchester, UK.

[Vou99] VOUDOURIS, C., TSANG, E. (1999). “Guided Local Search and its Application to the Travelling Salesman Problem”. *European Journal of Operations Research* **113**, 469-499.

[Wol98] WOLSEY, A. L. (1998), “*Integer Programming”* , Wiley, New York.

APÊNDICE

Médias, covariâncias e demais coeficientes estimados para o modelo de regressão utilizado no critério de parada automática

e

Código em *Object Pascal* do algoritmo proposto

Médias, covariâncias e demais coeficientes estimados para o modelo de regressão utilizado no critério de parada automática

2.2.1 Variáveis

$x_0 \equiv n$; $x_1 \equiv a$; $x_2 \equiv aproximação$; $x_3 \equiv n \times aproximação$; $x_4 \equiv a \times aproximação$; $x_5 \equiv n^2$; $x_6 \equiv a^2$ e $x_7 \equiv aproximação^2$

2.2.2 Variância

$s^2 = 0{,}19348451495171$

2.2.3 Coeficientes da regressão

$b_{-1} = 8{,}37914160669422$
$b_0 = 0{,}0079303162317396$
$b_1 = -4{,}89692161242129$
$b_2 = -1{,}28525509021886$
$b_3 = -0{,}000160445727116437$
$b_4 = -0{,}53791360344216$
$b_5 = -0{,}00000403456582016638$
$b_6 = 6{,}46567682022229$
$b_7 = 0{,}15512246462355$

Estimativas das médias das variáveis

$\bar{x}_0 = 500$; $\bar{x}_1 = 0{,}3$; $\bar{x}_2 = 2{,}25$; $\bar{x}_3 = 1.125$; $\bar{x}_4 = 0{,}675$; $\bar{x}_5 = 300.000$; $\bar{x}_6 = 0{,}11$ e $\bar{x}_7 = 7{,}90178571428571$

2.2.4 Matriz de covariâncias

$Cov(x_i, x_j)$	0	1	2	3
0	1,88138912449176E-07			
1	-3,86772826474209E-19	3,93841922283173E-01		
2	2,19039065996185E-06	3,28558590263128E-03	2,57824175059795E-03	
3	-4,38078107123374E-09	2,01321850890184E-20	-9,73506985246786E-07	1,94701388345209E-09
4	2,18928584098831E+20	-1,09519530087709E-02	-1,46026047877967E-03	-1,13772570181432E+20
5	-1,7275401942296E-10	5,38943156637166E-22	-1,53895857395466E-22	2,64711274691611E-25
6	5,54372000437247E-19	-5,9229952096939E-01	3,71207080466734E-16	8,78328404845578E-21
7	-1,2880509147442E-20	2,54855736391017E-17	-3,07155103655532E-04	3,66206266550437E-21

$Cov(x_i, x_j)$	4	5	6	7
0				
1				
2				
3				
4	4,86753461882472E-03			
5	8,42949942048396E-25	1,72754023000828E-13		
6	-1,25838623951712E-15	-8,78846079352955E-22	9,87165868282318E-01	
7	-1,14256845009755E-17	3,92194741919019E-24	4,49848217464127E-19	6,06302455707919E-05

Código em Object Pascal do algoritmo proposto

program SCR7;

```
uses
  Forms,
  UCityList in 'UCityList.pas',
  UPrincipal in 'UPrincipal.pas' {frmPrincipal},
  UArvMin in 'UArvMin.pas';
{$R *.RES}
begin
  Application.Initialize;
  Application.CreateForm(TfrmPrincipal, frmPrincipal);
  Application.Run;
end.
```

unit UPrincipal;

```
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, ComCtrls, StdCtrls, Buttons, Mask;
type
  TfrmPrincipal = class(TForm)
    SaveDialog1     : TSaveDialog;
    Panel6          : TPanel;
    OpenDialog1     : TOpenDialog;
    ScrollBox1      : TScrollBox;
    PaintBox1       : TPaintBox;
    SaveDialog2     : TSaveDialog;
    Panel2: TPanel;
    Panel4: TPanel;
    Memo1: TMemo;
    Panel1: TPanel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    BitBtn6: TBitBtn;
    Panel9: TPanel;
    Label4: TLabel;
    Label6: TLabel;
    Label8: TLabel;
    Label14: TLabel;
    Label15: TLabel;
    Bevel1: TBevel;
    Label5: TLabel;
    Label3: TLabel;
    Label17: TLabel;
    Label18: TLabel;
    Label2: TLabel;
    Label9: TLabel;
    edTemCPU: TEdit;
    edIter: TEdit;
    edTamRota: TEdit;
    edQtdClustVizinhos: TEdit;
    edMaxCidCluster: TEdit;
    chkAutoAjuste: TCheckBox;
    edMaxIter: TEdit;
    edParametro: TEdit;
    edAproximacao: TEdit;
    Panel8: TPanel;
    Label11: TLabel;
    Edit1: TEdit;
    btAbrir: TBitBtn;
    rgTipoDist: TRadioGroup;
    edSaida: TEdit;
    btOtimizar: TBitBtn;
    BitBtn4: TBitBtn;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    btSalvar: TBitBtn;
```

```
    edNumViz: TEdit;
    Label10: TLabel;
    Label12: TLabel;
    edNumTotClus: TEdit;
    Label13: TLabel;
    Label7: TLabel;
    edtNumCidades: TEdit;
    edNumCidClus: TEdit;
    Label16: TLabel;
    edClusID: TEdit;
    edNumClusResol: TEdit;
    Label19: TLabel;
    Label20: TLabel;
    edTemCPUFinal: TEdit;
    edTamRotaFinal: TEdit;
    SpeedButton1: TSpeedButton;
    SpeedButton2: TSpeedButton;
    Label21: TLabel;
    Label1: TLabel;
    chkClusters: TCheckBox;
    chkRotaComPart: TCheckBox;
    cbMetodo: TComboBox;
    Label22: TLabel;
    chkRotaSemPart: TCheckBox;
    Label23: TLabel;
    Label24: TLabel;
    edMenorMaior: TEdit;
    edMinCidCluster: TEdit;
    Label25: TLabel;
    procedure FormCreate(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
    procedure PaintBox1Paint(Sender: TObject);
    procedure btAbrirClick(Sender: TObject);
    procedure btOtimizarClick(Sender: TObject);
    procedure btGeraCidAleatClick(Sender: TObject);
    procedure BitBtn4Click(Sender: TObject);
    procedure SpeedButton1Click(Sender: TObject);
    procedure SpeedButton2Click(Sender: TObject);
    procedure btSalvarClick(Sender: TObject);
    procedure chkAutoAjusteClick(Sender: TObject);
    procedure chkRefreshClick(Sender: TObject);
    procedure BitBtn6Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
End;
Var
  frmPrincipal  : TfrmPrincipal;
implementation
{$R *.DFM}
Uses UCityList,UArvMin;

procedure TfrmPrincipal.BitBtn1Click(Sender: TObject);
Begin
  Memo1.Clear;
End;

procedure TfrmPrincipal.PaintBox1Paint(Sender: TObject);
var
    UmaListaCid :PTCityList;
begin
  If (UmaListaCidade <> Nil) or (UmGrafo<>nil)
     then CalculaEscala (PaintBox1);
  If UmaListaCidade <> Nil
  then begin
        If (UmaListaCidade.ListCity<>nil) and (UmaListaCidade.RotaOtima.NRota=0) and chkRotaSemPart.Checked=True
        then UmaListaCidade.MostrarCity(PaintBox1,CheckBox1.Checked);
        If (UmaListaCidade.ListCity<>nil) and (UmaListaCidade.RotaOtima.NRota>0) and chkRotaSemPart.Checked=True
        then UmaListaCidade.MostrarTudo(PaintBox1,CheckBox1.Checked);
       end;
  If UmGrafo <> Nil
  then begin
```

```
        If (UmGrafo.Vertices.Count>0)  and (chkClusters.Checked=True)
        then begin
             UmGrafo.MostraArvore(PaintBox1,CheckBox1.Checked);
            end;
        If (UmGrafo.Vertices.Count>0)  and (UmGrafo.RotaGlobal<>nil) and (chkRotaComPart.Checked=True)
        then begin
              UmaListaCid:=UmGrafo.RotaGlobal;
              UmaListaCid.MostrarTudo(PaintBox1,CheckBox1.Checked);
            end;
       end;
end;

procedure TfrmPrincipal.FormCreate(Sender: TObject);
begin
  Randomize;
end;

procedure TfrmPrincipal.btAbrirClick(Sender: TObject);
begin
  If OpenDialog1.Execute
  then begin
        Case cbMetodo.ItemIndex of
        0,1,2 : begin
                Edit1.Text := OpenDialog1.FileName;
                New (UmaListaCidade);
                UmaListaCidade.Ler(OpenDialog1.FileName,PaintBox1,Memo1);
                edtNumCidades.Text := IntToStr (UmaListaCidade.ListCity.Count);
               end;
        3,4,5 : begin
                Edit1.Text := OpenDialog1.FileName;
                UmGrafo.Done;
                New(UmGrafo,Init);
                UmGrafo.LerCityList(OpenDialog1.FileName,strtoint(edMaxCidCluster.text),
                                strtoint(edMinCidCluster.text),edMenorMaior,
                                rgTipoDist.ItemIndex,PaintBox1,Memo1);
                edtNumCidades.Text := IntToStr (UmGrafo.TotCidNosVertices);
               end;
          else begin
                Application.messagebox('Selecione primeiro um método de otimização.','Atenção!',MB_OK);
               end;
        end;
       end;
end;

procedure TfrmPrincipal.btOtimizarClick(Sender: TObject);
Var
   UmaListaCid   : PTCityList;
   UmaRota       : PTRota;
   Tempo         : Real;
   Fail          : Boolean;
   T0            : TDateTime;
begin
  Fail:=True;
  T0:=Now;
  if (UmaListaCidade<>nil) or (UmGrafo<>nil)
  then begin
        btOtimizar.Enabled:=False;
        btAbrir.Enabled:=False;
        btSalvar.Enabled:=False;
        rgTipoDist.Enabled:=False;
        chkRotaSemPart.Enabled:=False;
        chkRotaComPart.Enabled:=False;
        chkClusters.Enabled:=False;
        cbMetodo.enabled:=False;
        edNumTotClus.Text:='';
        edNumClusResol.Text:='';
        edClusID.Text:='';
        edNumCidClus.Text:='';
        edTamRota.Text:='';
        edIter.Text:='';
        edTemCPU.text:='';
        edTamRotaFinal.text:='';
        edTemCPUFinal.text:='';
        Memo1.lines.add('');
```

```
    end;
if (UmaListaCidade<>nil)
then begin
      Case cbMetodo.ItemIndex of
      0: begin
          ComArquivo := CheckBox2.Checked;
          Terminar := False;
          Memo1.lines.add('GFLS (vizin. completa)');
          Memo1.lines.add('======================');
          Memo1.Lines.add('Problema:'+edit1.text);
          UmaListaCidade.GlsTSP(edParametro,edAproximacao,edMaxIter,
                        edIter,edTamRota,edTemCPU,PaintBox1,
                        Memo1, clRed,5,rgTipoDist.ItemIndex,
                        strtoint(edNumViz.text),cbMetodo.itemIndex,
                        chkAutoAjuste.Checked);
          Tempo:=(Now-T0)*3600*24;
          UmaRota:=UmaListaCidade.RotaOtima;
          edTamRotaFinal.Text:=IntToStr(UmaRota.Custo);
          edTemCPUFinal.Text:=FloatToStrF(Tempo,ffFixed,10,1);
          UmaRota:=UmaListaCidade.RotaOtima;
          Memo1.lines.add('Número de Cidades='+IntToStr(UmaRota.NRota));
          Memo1.lines.add('Tamanho da Rota='+IntToStr(UmaRota.Custo));
          Memo1.lines.add('Tempo='+FloatToStrF(Tempo,ffFixed,10,1));
          Memo1.lines.add('======================');
          PaintBox1.Refresh;
          Fail:=False;
        end;
      1: begin
          ComArquivo := CheckBox2.Checked;
          Terminar := False;
          Memo1.lines.add('GFLS (vizin. completa só na 1a iter)');
          Memo1.lines.add('======================');
          Memo1.Lines.add('Problema:'+edit1.text);
          UmaListaCidade.GlsTSP(edParametro,edAproximacao,edMaxIter,
                        edIter,edTamRota,edTemCPU,PaintBox1,
                        Memo1, clRed,5,rgTipoDist.ItemIndex,
                        strtoint(edNumViz.text),cbMetodo.itemIndex,
                        chkAutoAjuste.Checked);
          Tempo:=(Now-T0)*3600*24;
          UmaRota:=UmaListaCidade.RotaOtima;
          edTamRotaFinal.Text:=IntToStr(UmaRota.Custo);
          edTemCPUFinal.Text:=FloatToStrF(Tempo,ffFixed,10,1);
          UmaRota:=UmaListaCidade.RotaOtima;
          Memo1.lines.add('Número de Cidades='+IntToStr(UmaRota.NRota));
          Memo1.lines.add('Tamanho da Rota='+IntToStr(UmaRota.Custo));
          Memo1.lines.add('Tempo='+FloatToStrF(Tempo,ffFixed,10,1));
          Memo1.lines.add('======================');
          PaintBox1.Refresh;
          Fail:=False;
        end;
      2: begin
          ComArquivo := CheckBox2.Checked;
          Terminar := False;
          Memo1.lines.add('GFLS (vizin. restrita)');
          Memo1.lines.add('======================');
          Memo1.Lines.add('Problema:'+ edit1.text);
          UmaListaCidade.GlsTSP(edParametro,edAproximacao,edMaxIter,
                        edIter,edTamRota,edTemCPU,PaintBox1,
                        Memo1, clRed,5,rgTipoDist.ItemIndex,
                        strtoint(edNumViz.text),cbMetodo.itemIndex,
                        chkAutoAjuste.Checked);
          Tempo:=(Now-T0)*3600*24;
          UmaRota:=UmaListaCidade.RotaOtima;
          edTamRotaFinal.Text:=IntToStr(UmaRota.Custo);
          edTemCPUFinal.Text:=FloatToStrF(Tempo,ffFixed,10,1);
          UmaRota:=UmaListaCidade.RotaOtima;
          Memo1.lines.add('Número de Cidades='+IntToStr(UmaRota.NRota));
          Memo1.lines.add('Tamanho da Rota='+IntToStr(UmaRota.Custo));
          Memo1.lines.add('Tempo='+FloatToStrF(Tempo,ffFixed,10,1));
          Memo1.lines.add('======================');
          PaintBox1.Refresh;
          Fail:=False;
        end;
      end;
```

```
    end;
if (UmGrafo<>nil)
then begin
      Case cbMetodo.ItemIndex of
      3: begin
         ComArquivo := CheckBox2.Checked;
         Terminar := False;
         Memo1.lines.add('Part. + GFLS (vizin. completa)');
         Memo1.lines.add('==============================');
         Memo1.Lines.add('Problema:'+ edit1.text);
         Memo1.Lines.add('Menor vértice : '+ inttostr(UmGrafo.MenorVertice));
         Memo1.Lines.add('Maior vértice : '+ inttostr(UmGrafo.MaiorVertice));
         UmGrafo.CriaListaDeVerticesVizinhos(StrToInt(edQtdClustVizinhos.text),rgTipoDist.ItemIndex);
         UmGrafo.OtimizaRotas(edParametro,edAproximacao,edMaxIter,
                        edIter,edTamRota,edTemCPU,
                        edNumTotClus,edNumClusResol,
                        edClusID,edNumCidClus,rgTipoDist.ItemIndex,
                        PaintBox1,Memo1,strtoint(edNumViz.text),
                        False,cbMetodo.itemIndex,chkAutoAjuste.Checked);
         UmGrafo.ConstroiArcos(Memo1,rgTipoDist.ItemIndex);
         UmGrafo.GeraRotaUnica(Memo1,PaintBox1);
         Tempo:=(Now-T0)*3600*24;
         UmaListaCid:=Umgrafo.RotaGlobal;
         UmaRota:=UmaListaCid.RotaOtima;
         edTamRotaFinal.Text:=IntToStr(UmaRota.Custo);
         edTemCPUFinal.Text:=FloatToStrF(Tempo,ffFixed,10,1);
         Memo1.lines.add('Número de Clusters='+IntToStr(UmGrafo.Vertices.Count));
         Memo1.lines.add('Número de Cidades='+IntToStr(UmaRota.NRota));
         Memo1.lines.add('Tamanho da Rota='+IntToStr(UmaRota.Custo));
         Memo1.lines.add('Tempo='+FloatToStrF(Tempo,ffFixed,10,1));
         Memo1.lines.add('==============================');
         PaintBox1.Refresh;
         Fail:=False;
        end;
      4: begin
         ComArquivo := CheckBox2.Checked;
         Terminar := False;
         Memo1.lines.add('Part. + GFLS (vizin.completa só na 1a iter.)');
         Memo1.lines.add('==========================================');
         Memo1.Lines.add('Problema:'+ edit1.text);
         Memo1.Lines.add('Menor vértice : '+ inttostr(UmGrafo.MenorVertice));
         Memo1.Lines.add('Maior vértice : '+ inttostr(UmGrafo.MaiorVertice));
         UmGrafo.CriaListaDeVerticesVizinhos(StrToInt(edQtdClustVizinhos.text),rgTipoDist.ItemIndex);
         UmGrafo.OtimizaRotas(edParametro,edAproximacao,edMaxIter,
                        edIter,edTamRota,edTemCPU,
                        edNumTotClus,edNumClusResol,
                        edClusID,edNumCidClus,rgTipoDist.ItemIndex,
                        PaintBox1,Memo1,strtoint(edNumViz.text),
                        False,cbMetodo.itemIndex,
                        chkAutoAjuste.Checked);
         memo1.lines.add('Otimizou!!!!!!!!'+FloatToStrF((Now-t0)*3600*24,ffFixed,10,1));
         UmGrafo.ConstroiArcos(Memo1,rgTipoDist.ItemIndex);
         memo1.lines.add('Constuiu arcos !!!!!!!!'+FloatToStrF((Now-t0)*3600*24,ffFixed,10,1));
         UmGrafo.GeraRotaUnica(Memo1,PaintBox1);
         memo1.lines.add('Gerou Rota Única!!!!!!!!'+FloatToStrF((Now-t0)*3600*24,ffFixed,10,1));
         Tempo:=(Now-T0)*3600*24;
         UmaListaCid:=Umgrafo.RotaGlobal;
         UmaRota:=UmaListaCid.RotaOtima;
         edTamRotaFinal.Text:=IntToStr(UmaRota.Custo);
         edTemCPUFinal.Text:=FloatToStrF(Tempo,ffFixed,10,1);
         PaintBox1.Refresh;
         Memo1.lines.add('Número de Clusters='+IntToStr(UmGrafo.Vertices.Count));
         Memo1.lines.add('Número de Cidades='+IntToStr(UmaRota.NRota));
         Memo1.lines.add('Tamanho da Rota='+IntToStr(UmaRota.Custo));
         Memo1.lines.add('Tempo='+FloatToStrF(Tempo,ffFixed,10,1));
         Memo1.lines.add('==========================================');
         PaintBox1.Refresh;
         Fail:=False;
        end;
      5: begin
         ComArquivo := CheckBox2.Checked;
         Terminar := False;
         Memo1.lines.add('Part. + GFLS (vizin. restrita)');
         Memo1.lines.add('==============================');
```

```
            Memo1.Lines.add('Problema:'+ edit1.text);
            Memo1.Lines.add('Menor vértice : '+ inttostr(UmGrafo.MenorVertice));
            Memo1.Lines.add('Maior vértice : '+ inttostr(UmGrafo.MaiorVertice));
            UmGrafo.CriaListaDeVerticesVizinhos(StrToInt(edQtdClustVizinhos.text),rgTipoDist.ItemIndex);
            UmGrafo.OtimizaRotas(edParametro,edAproximacao,edMaxIter,
                        edIter,edTamRota,edTemCPU,
                        edNumTotClus,edNumClusResol,
                        edClusID,edNumCidClus,rgTipoDist.ItemIndex,
                        PaintBox1,Memo1,strtoint(edNumViz.text),
                        False,cbMetodo.itemIndex,
                        chkAutoAjuste.Checked);
            UmGrafo.ConstroiArcos(Memo1,rgTipoDist.ItemIndex);
            UmGrafo.GeraRotaUnica(Memo1,PaintBox1);
            Tempo:=(Now-T0)*3600*24;
            UmaListaCid:=Umgrafo.RotaGlobal;
            UmaRota:=UmaListaCid.RotaOtima;
            edTamRotaFinal.Text:=IntToStr(UmaRota.Custo);
            edTemCPUFinal.Text:=FloatToStrF(Tempo,ffFixed,10,1);
            Memo1.lines.add('Número de Clusters='+IntToStr(UmGrafo.Vertices.Count));
            Memo1.lines.add('Número de Cidades='+IntToStr(UmaRota.NRota));
            Memo1.lines.add('Tamanho da Rota='+IntToStr(UmaRota.Custo));
            Memo1.lines.add('Tempo='+FloatToStrF(Tempo,ffFixed,10,1));
            Memo1.lines.add('==============================');
            PaintBox1.Refresh;
            Fail:=False;
          end;
        end;
      end;
   if Fail then Application.messagebox('Carregue o problema primeiro.','Atenção!',MB_OK);
   cbMetodo.enabled:=True;
   btOtimizar.Enabled:=True;
   btAbrir.Enabled:=True;
   btSalvar.Enabled:=True;
   rgTipoDist.Enabled:=True;
   chkRotaSemPart.Enabled:=True;
   chkRotaComPart.Enabled:=True;
   chkClusters.Enabled:=True;
end;

procedure TfrmPrincipal.btGeraCidAleatClick(Sender: TObject);
begin
  UmaListaCidade.RotaCorrente.GerarRandom (UmaListaCidade.ListCity.Count);
  PaintBox1.Refresh;
end;

procedure TfrmPrincipal.BitBtn4Click(Sender: TObject);
begin
  Terminar := True;
  Application.ProcessMessages;
end;

procedure TfrmPrincipal.SpeedButton1Click(Sender: TObject);
begin
  PaintBox1.Height:=trunc(PaintBox1.Height*2);
  PaintBox1.Width:=trunc(PaintBox1.Width*2);
  CalculaEscala(PaintBox1);
end;

procedure TfrmPrincipal.SpeedButton2Click(Sender: TObject);
begin
  PaintBox1.Height:=trunc(PaintBox1.Height/2);
  PaintBox1.Width:=trunc(PaintBox1.Width/2);
end;

procedure TfrmPrincipal.btSalvarClick(Sender: TObject);
begin
 If SaveDialog2.Execute then
  begin
    edSaida.text:=SaveDialog2.Filename;
    UmaListaCidade.Gravar (SaveDialog1.Filename);
  end;
end;

procedure TfrmPrincipal.chkAutoAjusteClick(Sender: TObject);
```

```
begin
  if chkAutoAjuste.Checked=true
  then begin
         edParametro.enabled:=False;
         edMaxIter.enabled:=False;
       end
  else begin
         edParametro.enabled:=True;
         edMaxIter.enabled:=True;
       end;
end;

procedure TfrmPrincipal.BitBtn6Click(Sender: TObject);
begin
  Panel2.Height:=113;
  Panel4.Height:=113;
end;

procedure TfrmPrincipal.chkRefreshClick(Sender: TObject);
begin
  PaintBox1.Refresh;
end;

procedure TfrmPrincipal.BitBtn2Click(Sender: TObject);
begin
  Panel2.Height:=400;
  Panel4.Height:=400;
end;

End.
```

unit UCityList;

```
interface
Uses UPrincipal,Comctrls,extctrls,Windows,
     classes,SysUtils,graphics,Forms,stdctrls;
Type
  TArray_of_Array_of_Real = Array of Array of Real;
  PTRota = ^TRota;
  TRota = object
    No           : Array of Integer;
    NRota        : Integer;
    Custo        : Int64;
    CustoH       : Real;
    Cor          : TColor;
    LarguraLinha : Integer;
    Constructor Init (InitCor : TColor; LargLinha: Integer);
    Destructor Done;
    Procedure GerarRandom (InitN : Integer);
    Function GetNo (k : Integer) : Integer;
    Function GetSuc (k,p : Integer) : Integer;
    Function GetPos (k : Integer) : Integer;
    Procedure Inverter (A,B : Integer);
    Procedure Recebe (UmaRota : PTRota);
  end;
  TVizinProx = Record
                 IndiceNaLista : Integer;
                 Dist          : Int64;
               End;
  PTCity = ^TCity;
  TCity = object
    Id               : Integer;
    CoordX,CoordY    : Real;
    VizinhosProx     : Array of TVizinProx;
    Constructor Init (InitId : Integer; InitX,InitY : Real);
    Destructor Done;
    Procedure GeraRandom (InitId : Integer; XMin,XMax,YMin,Ymax: Real);
    Procedure Ler (Var Arq : TextFile);
    Function PontoTela (UmPaintBox : TPaintBox) : TPoint;
    Procedure Mostrar (UmPaintBox : TPaintBox; ComID : Boolean);
    Procedure Gravar (Var Arq : TextFile);
```

```
  end;
  PTCityList = ^TCityList;
  TCityList = object
    ListCity              : TList;
    RotaOtima,RotaCorrente : PTRota;
    Destructor Done;
    Procedure GeraRandom(InitN : Integer; XMin,XMax,YMin,Ymax: Real);
    Procedure Ler(FileName:String; UmPaintBox:TPaintBox; UmMemo:TMemo);
    Procedure Gravar(FileName : String);
    Procedure MostrarCity(UmPaintBox : TPaintBox; ComID : Boolean);
    Procedure MostrarRota(UmaRota : PTRota; UmPaintBox : TPaintBox);
    Procedure MostrarTudo(UmPaintBox : TPaintBox; ComID : Boolean);
    Function Dist(i,j: integer):Int64;
    Procedure CalcularDistancias(UmTipoDist : Integer);
    Procedure CalcularDistanciasB(UmTipoDist : Integer);
    Function AvaliaRota(UmaRota : PTRota) : Int64;
    Function AvaliaRotaH(UmaRota : PTRota) : Real;
    Procedure GerarListaCidadesVizinhas(k:Integer; UmMemo:TMemo);
    Procedure GlsTSP(edParametro,edAproximacao,
                     edMaxIter,edIter,
                     edTamRota,edTemCPU        : TEdit;
                     UmPaintBox                : TPaintBox;
                     UmMemo                    : TMemo;
                     CorRota                   : TColor;
                     UmaLargLinha              : Integer;
                     UmTipoDist                : Integer;
                     NVizinhos                 : Integer;
                     Variantes                 : Integer;
                     AutoAjuste                : Boolean);
    Procedure InicializarGLS( CorRota   : TColor ;
                              LargLinha : integer;
                              TipoDist : Integer;
                              NVizinhos : Integer;
                              Vizinhos  : Boolean;
                              UmMemo   : TMemo);
    Procedure FinalizarGLS;
    Procedure AjustaPenalidades;
    Procedure BuscaLocal(edOtimo,edCPU : TEdit);
    Procedure BuscaLocal_Lista(edOtimo,edCPU : TEdit);
    Function RetornaProximoBitUm(k: Integer) : Integer;
    Function Realiza2Opt(k : Integer;edOtimo,edCPU : TEdit) : Boolean;
    Function Realiza2Opt_Lista(k : Integer;edOtimo,edCPU : TEdit) : Boolean;
    Function Penal(i,j: integer):integer;
    Procedure IniciaPenalidades;
    Procedure AltPenal(i,j,valor:integer);
    Function DeterminaParametro(Aproximacao:Real):Real;
    Function DeterminaMaxIter(Aproximacao,a:Real; n:Integer):Int64;
  end;
Procedure CalculaEscala(UmPaintBox:TPaintBox);
Function DistEuclid_2d(x1,y1,x2,y2:Real):Int64;
Function DistCeil_2d(x1,y1,x2,y2:Real):Int64;
Procedure QuickSortReal(var A: TArray_of_Array_of_Real; iLo, iHi: Integer);

Var
  XMax,YMax,XMin,YMin,Escala : Real;
  UmaListaCidade             : PTCityList;
  Lambda                     : Real;
  Inicio                     : TDateTime;
  Terminar,
  ComArquivo                 : Boolean;
  Iter                       : Integer;
  Arq                        : TextFile;
  Distancias                 : Array of Array of Int64;
  Penalidades                : Array of Array of Integer;
  Bits                       : Array of Byte;
  ListaMaxUtil               : Array [0..2999,0..1] of Integer;

implementation

Procedure QuickSortReal(var A: TArray_of_Array_of_Real; iLo, iHi: Integer);
Var
  Lo, Hi  : Integer;
  T, Mid  : Real;
begin
```

```
  Lo := iLo;
  Hi := iHi;
  Mid := A[(Lo + Hi) div 2,1];
  repeat
    while A[Lo,1] < Mid do Inc(Lo);
      while A[Hi,1] > Mid do Dec(Hi);
        if Lo <= Hi
        then begin
               T := A[Lo,1];
               A[Lo,1] := A[Hi,1];
               A[Hi,1]:= T;
               T:= A[Lo,0];
               A[Lo,0] := A[Hi,0];
               A[Hi,0] := T;
               Inc(Lo);
               Dec(Hi);
             end;
  until Lo > Hi;
  if Hi > iLo then QuickSortReal(A, iLo, Hi);
  if Lo < iHi then QuickSortReal(A, Lo, iHi);
end;

Procedure CalculaEscala (UmPaintBox : TPaintBox);
Var
  Esc1,Esc2 : Real;
begin
  Esc1 := (UmPaintBox.Width - 20) / (XMax - XMin);
  Esc2 := (UmPaintBox.Height - 20) / (YMax - YMin);
  If Esc1 > Esc2
  then Escala := Esc2
  else Escala := Esc1;
end;

Function DistEuclid_2d(x1,y1,x2,y2:Real):Int64;
begin
  Result:=round(sqrt(sqr(x1-x2)+sqr(y1-y2)));
end;

Function DistCeil_2d(x1,y1,x2,y2:Real):Int64;
begin
  Result:=round(sqrt(sqr(x1-x2)+sqr(y1-y2))+0.5);
end;

Constructor TRota.Init(InitCor : TColor; LargLinha: Integer);
begin
  NRota := 0;
  Cor := InitCor;
  LarguraLinha :=LargLinha;
end;

Procedure TRota.GerarRandom (InitN : Integer);
Var
  Aux,i,j,k : Integer;
begin
  Randomize;
  NRota := InitN;
  Finalize(No);
  SetLength (No,NRota);
  For i := 0 to NRota-1 do No [i] := i;
  For k := 0 to NRota-1 do
  begin
    i := Random (NRota);
    j := Random (NRota);
    Aux := No [i];
    No [i] := No [j];
    No [j] := Aux;
  end;
end;

Function TRota.GetNo (k : Integer) : Integer;
begin
  Result := No [k];
end;
```

```
Function TRota.GetSuc (k,p : Integer) : Integer;
begin
  k := (k + p + NRota) mod NRota;
  Result := No [k];
end;

Function TRota.GetPos (k : Integer) : Integer;
Var
  i : Integer;
begin
  i := 0;
  While No [i] <> k do i := i + 1;
  Result := i;
end;

Procedure TRota.Recebe (UmaRota : PTRota);
Var
  i : Integer;
begin
  If NRota <> UmaRota.NRota
  then begin
         No := nil;
         NRota := UmaRota.NRota;
         SetLength (No,NRota);
       end;
  For i := 0 to NRota - 1 do No [i] := UmaRota.No[i];
  Custo := UmaRota.Custo;
end;

Procedure TRota.Inverter (A,B : Integer);
Var
  PosA,PosB,Aux,N,k : Integer;

begin
  PosA:= (A + NRota) mod NRota;
  PosB:= (B + NRota) mod NRota;
  If PosA <= PosB
  then N := PosB - PosA + 1
  else N := NRota - PosA + PosB + 1;
  N := N div 2 - 1;
  For k := 0 to N do
  begin
    Aux := No [PosA];
    No [PosA] := No [PosB];
    No [PosB] := Aux;
    PosA := PosA + 1;
    If PosA > NRota - 1 then PosA := 0;
    PosB := PosB - 1;
    If PosB < 0 then PosB := NRota - 1;
  end;
end;

Destructor TRota.Done;
begin
  Finalize(No);
end;

Procedure TCityList.GeraRandom (InitN : Integer; XMin,XMax,YMin,Ymax: Real);
Var
  UmaCity : PTCity;
  i       : Integer;
begin
  ListCity := TList.Create;
  For i := 1 to InitN do
  begin
    New (UmaCity);
    UmaCity.GeraRandom(i,XMin,XMax,YMin,YMax);
    ListCity.Add (UmaCity);
  end;
  New (RotaCorrente,Init (clSilver,1));
  New (RotaOtima,Init (clRed,1));
  Xmax := -1E+30;
  Ymax := -1E+30;
  XMin := +1E+30;
```

```
  YMin := +1E+30;
  For i := 0 to ListCity.Count-1 do
  begin
    UmaCity := ListCity.Items [i];
    If XMax < UmaCity.CoordX then XMax := UmaCity.CoordX;
    If YMax < UmaCity.CoordY then YMax := UmaCity.CoordY;
    If XMin > UmaCity.CoordX then XMin := UmaCity.CoordX;
    If YMin > UmaCity.CoordY then YMin := UmaCity.CoordY;
  end;
end;

Procedure TCityList.Ler (FileName : String; UmPaintBox: TPaintBox; UmMemo:TMemo);
Var
  Arq      : TextFile;
  UmaCity   : PTCity;
  i        : Integer;
begin
  ListCity := TList.Create;
  AssignFile (Arq,FileName);
  Reset (Arq);
  While not EOF(Arq) do
  begin
    New (UmaCity);
    UmaCity.Ler (Arq);
    ListCity.Add (UmaCity);
  end;
  Close (Arq);
  Xmax := -1E+30;
  Ymax := -1E+30;
  XMin := +1E+30;
  YMin := +1E+30;
  For i := 0 to ListCity.Count-1 do
  begin
    UmaCity := ListCity.Items [i];
    If XMax < UmaCity.CoordX then XMax := UmaCity.CoordX;
    If YMax < UmaCity.CoordY then YMax := UmaCity.CoordY;
    If XMin > UmaCity.CoordX then XMin := UmaCity.CoordX;
    If YMin > UmaCity.CoordY then YMin := UmaCity.CoordY;
  end;
  New (RotaCorrente,Init (clSilver,1));
  New (RotaOtima,Init (clRed,1));
  UmPaintBox.Refresh;
end;

Destructor TCityList.Done;
Var
  UmaCity : PTCity;
  i             : Integer;
begin
  For i := 0 to ListCity.Count-1 do
  begin
    UmaCity := ListCity.Items[i];
    Dispose (UmaCity,Done);
  end;
  ListCity.Free;
end;

Function TCityList.AvaliaRota (UmaRota : PTRota) : Int64;
Var
  i,Inicio,Fim : Integer;
begin
  Result := 0;
  For i := 0 to UmaRota.NRota - 1 do
  begin
    Inicio := UmaRota.GetNo (i);
    Fim := UmaRota.GetSuc (i,1);
    Result := Result + Dist (Inicio,Fim);
  end;
end;

Function TCityList.AvaliaRotaH (UmaRota : PTRota) : Real;
Var
  i,Inicio,Fim : Integer;
  Soma         : Real;
```

```
begin
  Result := 0;
  Soma := 0;
  For i := 0 to UmaRota.NRota - 1 do
  begin
    Inicio := UmaRota.GetNo (i);
    Fim := UmaRota.GetSuc (i,1);
    Result := Result + Dist (Inicio,Fim);
    Soma := Soma + Penal (Inicio,Fim);
  end;
  Result := Result + Lambda * Soma;
end;

Procedure TCityList.InicializarGLS( CorRota   : TColor;
                                    LargLinha : Integer;
                                    TipoDist  : Integer;
                                    NVizinhos : Integer;
                                    Vizinhos  : Boolean;
                                      UmMemo : TMemo);
Var
  i,j : Integer;
begin
  CalcularDistancias(TipoDist);
  IF Vizinhos
  then if NVizinhos>0
       then GerarListaCidadesVizinhas(NVizinhos,UmMemo)
       else GerarListaCidadesVizinhas(40,UmMemo);
  IniciaPenalidades;
  For i := 0 to ListCity.Count-1 do
    For j := 0 to ListCity.Count-1 do AltPenal(i,j,0);
  Bits := nil;
  SetLength (Bits,ListCity.Count);
  For i := 0 to ListCity.Count-1 do Bits [i] := 1;
  RotaCorrente.GerarRandom(ListCity.Count);
  RotaCorrente.Custo := AvaliaRota (RotaCorrente);
  RotaOtima.Recebe (RotaCorrente);
  Inicio := Now;
end;

Procedure TCityList.FinalizarGLS;
begin
    FINALIZE(Distancias);
    FINALIZE(Penalidades);
    FINALIZE(Bits);
    RotaCorrente.Done;
end;

Procedure TCityList.AjustaPenalidades;
Var
  i,j,k       : Integer;
  MaxUtil,Util : Real;
  Pont     : Integer;

begin
  MaxUtil := -1E+30;
  Pont:=-1;
  For k := 0 to RotaCorrente.NRota - 1 do
  begin
    i := RotaCorrente.GetNo(k);
    j := RotaCorrente.GetSuc (k,1);
    Util := Dist (i,j) / (1 + Penal(i,j));
    If (Util >= MaxUtil)
    then Begin
           Pont:=Pont+1;
           If Util > MaxUtil
           then Begin
                  MaxUtil := Util;
                  Pont := 0;
                End;
           ListaMaxUtil[Pont,0]:=i;
           ListaMaxUtil[Pont,1]:=j;
         End;
  end;
  For k := 0 to Pont do
```

```
  begin
     AltPenal(ListaMaxUtil[k,0],ListaMaxUtil[k,1],
           Penal(ListaMaxUtil[k,0],ListaMaxUtil[k,1])+1);
     Bits [ListaMaxUtil[k,0]] := 1;
     Bits [ListaMaxUtil[k,1]] := 1;
  end;
end;

Procedure TCityList.BuscaLocal (edOtimo,edCPU : TEdit);
Const
  NaoTem = -1;
Var
  k : Integer;
begin
  k :=   RetornaProximoBitUm (-1);
  While k <> NaoTem do
  begin
    If not Realiza2Opt (k,edOtimo,edCPU)
    then Bits [k] := 0;
    k := RetornaProximoBitUm (k);
  end;
end;

Procedure TCityList.BuscaLocal_Lista(edOtimo,edCPU : TEdit);
Const
  NaoTem = -1;
Var
  k : Integer;
begin
  k :=   RetornaProximoBitUm (-1);
  While k <> NaoTem do
  begin
    If not Realiza2Opt_Lista (k,edOtimo,edCPU)
    then Bits [k] := 0;
    k := RetornaProximoBitUm (k);
  end;
end;

Function TCityList.Realiza2Opt (k : Integer;edOtimo,edCPU : TEdit) : Boolean;
Var
  t,a,d,i0,i1,j0,j1 : Integer;
  Delta_g           : Int64;
  Delta_h           : Real;
begin
  Result := False;
  t := RotaCorrente.GetPos (k);
  For a := -1 to 0 do
  begin
    i0 := RotaCorrente.GetSuc (t,a);
    j0 := RotaCorrente.GetSuc (t,a+1);
    For d:= 0 to ListCity.Count - 4 do
    begin
      i1 := RotaCorrente.GetSuc (t+2,a+d);
      j1 := RotaCorrente.GetSuc (t+3,a+d);
      Delta_g := Dist (i0,j0) + Dist (i1,j1) -
              Dist (i0,i1) - Dist (j0,j1);
      Delta_h := Delta_g + Lambda *
              (Penal(i0,j0) + Penal(i1,j1) - Penal(i0,i1) - Penal(j0,j1));
      If Delta_h > 0 then
      begin
        Bits [i0] := 1;
        Bits [i1] := 1;
        Bits [j0] := 1;
        Bits [j1] := 1;
        RotaCorrente.Inverter (t+a+1,t+2+a+d);
        RotaCorrente.Custo := RotaCorrente.Custo - Delta_g;
        Result := True;
        If RotaCorrente.Custo < RotaOtima.Custo  then RotaOtima.Recebe(RotaCorrente);
        Exit;
      end;
    end;
  end;
end;
```

```
{******************    Busca 2-opt com lista de Vizinhos    ******************}

Function TCityList.Realiza2Opt_Lista (k : Integer;edOtimo,edCPU : TEdit) : Boolean;
Var
  t,a,
  i0,i1,j0,j1,TamVizin,
  IndCandidato,PosCandidato,
  Candidato,Sucj0             : Integer;
  Delta_g                     : Int64;
  Delta_h                     : Real;
  d0                          : Int64;
  UmaCity                     : PTCity;
  UmVizinho                   : TVizinPRox;
begin
  Result := False;
  t := RotaCorrente.GetPos (k);
  For a := -1 to 0 do
  begin
    i0 := RotaCorrente.GetSuc (t,a);
    j0 := RotaCorrente.GetSuc (t,a+1);
    UmaCity:=ListCity.items[j0];
    TamVizin:=High(UmaCity.VizinhosProx);
    For IndCandidato:= 0 to TamVizin do
    begin
      d0:=Dist(i0,j0);
      UmVizinho:=UmaCity.VizinhosProx[IndCandidato];
      Candidato:=UmVizinho.IndiceNaLista;
      Sucj0:=RotaCorrente.GetSuc(j0,1);
      if (UmVizinho.Dist<d0) and (Sucj0<>Candidato) and (Candidato<>i0)
      then begin
            PosCandidato:= RotaCorrente.GetPos(Candidato);;
            i1 :=RotaCorrente.GetNo(PosCandidato);
            j1:= RotaCorrente.GetSuc (PosCandidato,1);
            Delta_g := Dist (i0,j0) + Dist (i1,j1) -
               Dist (i0,i1) - Dist (j0,j1);
            Delta_h := Delta_g + Lambda *
                  (Penal(i0,j0) + Penal(i1,j1) - Penal(i0,i1) - Penal(j0,j1));
            If Delta_h > 0
            then begin
                  Bits [i0] := 1;
                  Bits [i1] := 1;
                  Bits [j0] := 1;
                  Bits [j1] := 1;
                  RotaCorrente.Inverter (t+a+1,PosCandidato);
                  RotaCorrente.Custo := RotaCorrente.Custo - Delta_g;
                  Result := True;
                  If RotaCorrente.Custo < RotaOtima.Custo
                  then RotaOtima.Recebe(RotaCorrente);
                  Exit;
                end;
          end;
      end;
  end;
end;

Procedure TCityList.Gravar (FileName : String);
Var
  Arq       : TextFile;
  UmaCity   : PTCity;
  k         : Integer;
begin
  AssignFile (Arq,FileName);
  Rewrite (Arq);
  For k := 0 to ListCity.Count -1 do
  begin
    UmaCity := ListCity.Items [k];
    UmaCity.Gravar (Arq);
  end;
  Close (Arq);
end;

Function TCityList.Dist(i,j: integer):Int64;
begin
 if i>=j then
```

```
      Result:=Distancias[i,j]
      else
      Result:=Distancias[j,i];
end;

Function TCityList.Penal(i,j: integer):Integer;
begin
 if i>=j then
      Result:=Penalidades[i,j]
      else
      Result:=Penalidades[j,i];
end;

Procedure TCityList.CalcularDistancias(UmTipoDist : Integer);
Var
  N,i,j       : Integer;
  Cityi,Cityj  : PTCity;
begin
  N := ListCity.Count;
  Distancias := Nil;
  SetLength (Distancias,N);
  Case UmTipoDist of
  0 : begin
      For i := 0 to N-1 do
      begin
         SetLength(Distancias[i],i+1);
         Cityi := ListCity.Items [i];
         For j := 0 to i do
         begin
            Cityj := ListCity.Items [j];
            Distancias [i,j] := DistEuclid_2D(Cityi.CoordX,Cityi.CoordY,Cityj.CoordX,Cityj.CoordY);
         end;
      end;
     end;
  1 : begin
      For i := 0 to N-1 do
      begin
         SetLength(Distancias[i],i+1);
         Cityi := ListCity.Items [i];
         For j := 0 to i do
         begin
            Cityj := ListCity.Items [j];
            Distancias [i,j] :=DistCeil_2D(Cityi.CoordX,Cityi.CoordY,Cityj.CoordX,Cityj.CoordY);
         end;
      end;
     end;
  end;
end;

Procedure TCityList.CalcularDistanciasB(UmTipoDist : Integer);
Var
  N,i,j       : Integer;
  Cityi,Cityj  : PTCity;
begin
  N := ListCity.Count;
  Distancias := Nil;
  SetLength (Distancias,N);
  Case UmTipoDist of
  0 : begin
      For i := 0 to N-1 do
      begin
         SetLength(Distancias[i],i+1);
         Cityi := ListCity.Items [i];
         For j := 0 to i do
         begin
            Cityj := ListCity.Items [j];
            Distancias [i,j] := DistEuclid_2D(Cityi.CoordX,Cityi.CoordY,Cityj.CoordX,Cityj.CoordY);
         end;
      end;
     end;
  1 : begin
      For i := 0 to N-1 do
      begin
         SetLength(Distancias[i],i+1);
```

```
          Cityi := ListCity.Items [i];
          For j := 0 to i do
          begin
            Cityj := ListCity.Items [j];
            Distancias [i,j] := DistCeil_2D(Cityi.CoordX,Cityi.CoordY,Cityj.CoordX,Cityj.CoordY);
          end;
       end;
     end;
  end;
end;

Procedure TCityList.IniciaPenalidades;
Var
  N,i,j       : Integer;
begin
  N := ListCity.Count;
  Penalidades := nil;
  SetLength (Penalidades,N);
  For i := 0 to N-1 do
  begin
    SetLength(Penalidades[i],i+1);
    For j := 0 to i do
    begin
      Penalidades [i,j] :=0;
    end;
  end;
end;

Procedure TCityList.AltPenal(i,j,valor:integer);
begin
 if i>=j then
          Penalidades[i,j]:=valor
         else
          Penalidades[j,i]:=valor;
end;

Procedure TCityList.GlsTSP (edParametro,edAproximacao,
                        edMaxIter,edIter,
                        edTamRota,edTemCPU         : TEdit;
                        UmPaintBox                 : TPaintBox;
                        UmMemo                     : TMemo;
                        CorRota                  : TColor;
                        UmaLargLinha               : Integer;
                        UmTipoDist                 : Integer;
                        NVizinhos                  : Integer;
                        Variantes                  : Integer;
                        AutoAjuste                 : Boolean);
Var
  Otimo : Real;
  MaxIter:Int64;
begin
  Case Variantes of
     0,3,6: InicializarGLS(CorRota,UmaLargLinha,UmTipoDist,NVizinhos,False,UmMemo);
  1,2,4,5,7: InicializarGLS(CorRota, UmaLargLinha,UmTipoDist,NVizinhos,True,UmMemo);
  end;
  if Autoajuste=true
  then begin
       Lambda:=DeterminaParametro(StrToFloat(edAproximacao.text));
       MaxIter:=DeterminaMaxIter(StrToFloat(edAproximacao.text),Lambda,ListCity.Count);
       edParametro.Text:=FloatToStrF(Lambda,ffFixed,3,5);
       edMaxIter.Text:=IntToStr(MaxIter);
      end
  else begin
       Lambda := StrToFloat(edParametro.Text);
       MaxIter:=StrToInt(edMaxIter.Text);
  end;
  Iter := 0;
  Otimo := RotaOtima.Custo;
  While (Iter < MaxIter) and not Terminar do
  begin
    Iter := Iter + 1;
    Case Variantes of
    0,3: BuscaLocal (edTamRota,edTemCPU);
    1,4,7: begin
```

```
          if iter=1
          then BuscaLocal(edTamRota,edTemCPU)
          else BuscaLocal_Lista(edTamRota,edTemCPU);
        end;
    2,5: BuscaLocal_Lista (edTamRota,edTemCPU);
    end;
    If (RotaOtima.Custo<Otimo)
    then begin
          Otimo := RotaOtima.Custo;
          edTemCPU.Text := FloatToStrF ((Now-Inicio)*24*3600,ffFixed,10,2);
          edIter.Text := IntToStr(Iter);
          edTamRota.Text:=IntToStr(RotaOtima.Custo);
          Application.ProcessMessages;
        end;
    If Iter = 1 then Lambda := Lambda * RotaOtima.Custo / RotaOtima.NRota;
    AjustaPenalidades;
  end;
  edTemCPU.Text := FloatToStrF ((Now-Inicio)*24*3600,ffFixed,10,2);
  edIter.Text := IntToStr(Iter);
  edTamRota.Text:=IntToStr(RotaOtima.Custo);
  FinalizarGLS;
end;

Procedure TCityList.MostrarCity (UmPaintBox : TPaintBox; ComID : Boolean);
Var
  UmaCity : PTCity;
  i       : Integer;
begin
  For i := 0 to ListCity.Count-1 do
  begin
    UmaCity:=ListCity.Items[i];
    UmaCity.Mostrar (UmPaintBox,ComID);
  end;
end;

Procedure TCityList.MostrarRota (UmaRota : PTRota; UmPaintBox : TPaintBox);
Var
  UmPto            : TPoint;
  UmaCity          : PTCity;
  i,N              : Integer;
begin
  if UmaRota<>nil then
  begin
    N := UmaRota.NRota - 1;
    If N>=0 then With UmPaintBox.Canvas do
    begin
      Pen.Color := clBlack;
      Pen.Color := UmaRota.Cor;
      Pen.Width := UmaRota.LarguraLinha;
      UmaCity := ListCity.Items [UmaRota.No[N]];
      UmPto := UmaCity.PontoTela (UmPaintBox);
      MoveTo (UmPto.X,UmPto.Y);
      For i := 0 to N do
      begin
        UmaCity := ListCity.Items [UmaRota.No[i]];
        UmPto := UmaCity.PontoTela (UmPaintBox);
        LineTo (UmPto.X,UmPto.Y);
      end;
    end;
  end;
end;

Procedure TCityList.MostrarTudo (UmPaintBox : TPaintBox; ComID : Boolean);
begin
  MostrarRota (RotaOtima,UmPaintBox);
  MostrarCity (UmPaintBox,ComID);
end;

Function TCityList.RetornaProximoBitUm (k : Integer) : Integer;
Var
  i,j : Integer;
begin
  Result := -1;
  For i := 1 to ListCity.Count do
```

```
  begin
    j := (i + k) mod ListCity.Count;
    If Bits [j] = 1 then
    begin
      Result := j;
      exit;
    end;
  end;
end;

Function TCityList.DeterminaParametro(Aproximacao:Real):Real;
begin
    Result:=(0.53791360344216*Aproximacao+4.89692161242129)/(2*6.46567682022229);
end;

Function TCityList.DeterminaMaxIter(Aproximacao,a:Real; n:Integer):Int64;
Var
  Cov      : array [0..7,0..7] of Real;
  Med,X    : array [0..7] of Real;
  Beta     : array [-1..7] of Real;
  i,j      : Integer;
  Soma,V,Y : Real;
begin
  V:=0.19348451495171;
  X[0]:=n;
  X[1]:=a;
  X[2]:=aproximacao;
  X[3]:=n*aproximacao;
  X[4]:=aproximacao*a;
  X[5]:=n*n;
  X[6]:=a*a;
  X[7]:=aproximacao*aproximacao;
  Beta[-1]:=8.37914160669422;
  Beta[0]:=0.0079303162317396;
  Beta[1]:=-4.89692161242129;
  Beta[2]:=-1.28525509021886;
  Beta[3]:=-0.000160445727116437;
  Beta[4]:=-0.53791360344216;
  Beta[5]:=-0.00000403456582016638;
  Beta[6]:=6.46567682022229;
  Beta[7]:=0.15512246462355;
  Med[0]:=500;
  Med[1]:=0.3;
  Med[2]:=2.25;
  Med[3]:=1125;
  Med[4]:=0.675;
  Med[5]:=300000;
  Med[6]:=0.11;
  Med[7]:=7.90178571428571;
  Cov[0,0]:=0.000000188138912449176;
  Cov[1,0]:=-0.00000000000000000386772826474209;
  Cov[1,1]:=0.393841922283173;
  Cov[2,0]:=0.00000219039065996185;
  Cov[2,1]:=0.00328558590263128;
  Cov[2,2]:=0.00257824175059795;
  Cov[3,0]:=-0.0000000043807810712337 4;
  Cov[3,1]:=0.00000000000000000000201321850890184;
  Cov[3,2]:=-0.000000973506985246786;
  Cov[3,3]:=0.0000000019470138834 5209;
  Cov[4,0]:=0.00000000000000000000218928584098831;
  Cov[4,1]:=-0.0109519530087709;
  Cov[4,2]:=-0.00146026047877967;
  Cov[4,3]:=-0.00000000000000000000113772570181432;
  Cov[4,4]:=0.00486753461882472;
  Cov[5,0]:=-0.00000000017275401942296;
  Cov[5,1]:=0.00000000000000000000000538943156637166;
  Cov[5,2]:=-0.00000000000000000000000153895857395466;
  Cov[5,3]:=0.00000000000000000000000000264711274691611;
  Cov[5,4]:=0.00000000000000000000000000842949942048396;
  Cov[5,5]:=0.00000000000017275402300 0828;
  Cov[6,0]:=0.00000000000000000000554372000437247;
  Cov[6,1]:=-0.59229952096939;
  Cov[6,2]:=0.00000000000000003712070804 66734;
  Cov[6,3]:=0.00000000000000000000000878328404845578;
```

```
  Cov[6,4]:=-0.00000000000000125838623951712;
  Cov[6,5]:=-0.00000000000000000000000878846079352955;
  Cov[6,6]:=0.987165868282318;
  Cov[7,0]:=-0.00000000000000000000012880509147442;
  Cov[7,1]:=0.00000000000000000254855736391017;
  Cov[7,2]:=-0.000307155103655532;
  Cov[7,3]:=0.00000000000000000000366206266550437;
  Cov[7,4]:=-0.00000000000000000114256845009755;
  Cov[7,5]:=0.00000000000000000000000392194741919019;
  Cov[7,6]:=0.00000000000000000000449848217464127;
  Cov[7,7]:=0.000060630245570791 9;
  For j:=0 to 6 do
    For i:= j+1 to 7 do
      Cov[j,i]:=Cov[i,j];
  Y:=Beta[-1];
  For i:=0 to 7 do
    Y:=Y+Beta[i]*X[i];
  Soma:=0;
  For i:=0 to 7 do
    For j:=0 to 7 do
      Soma:=Soma+(X[i]-Med[i])*(X[j]-Med[j])*Cov[i,j];
  Result:=Trunc(Exp(Y+1.96*(V *(1+1/700)+Soma)));
end;

Procedure TCityList.GerarListaCidadesVizinhas(k:Integer; UmMemo:TMemo);
Var
   i,j,m,minimo  : integer;
   UmaCity       : PTCity;
   Lista         : TArray_of_Array_of_Real;
begin
  Finalize(Lista);
  SetLength(Lista,ListCity.Count-1,2);
  For i:=0 to ListCity.Count-1 do
  begin
    m:=0;
    For j:=0 to ListCity.Count-1 do
    begin
      if i<>j
      then begin
             Lista[m,0]:=j;
             Lista[m,1]:=Dist(i,j);
             m:=m+1;
           end;
    end;
    QuickSortReal(Lista,0,ListCity.Count-2);
    minimo:=ListCity.Count-1;
    if k<minimo then minimo:=k;
    UmaCity:=ListCity.Items[i];
    SetLength(UmaCity.VizinhosProx,minimo);
    m:=0;
    While m<=minimo-1 do
    begin
      UmaCity.VizinhosProx[m].IndiceNaLista:=trunc(Lista[m,0]);
      UmaCity.VizinhosProx[m].Dist:=Trunc(Lista[m,1]);
      m:=m+1;
    end;
  end;
end;

Constructor TCity.Init (InitId : Integer; InitX,InitY : Real);
begin
  Id     := InitId;
  CoordX := InitX;
  CoordY := InitY;
end;

Procedure TCity.GeraRandom (InitId : Integer; XMin,XMax,YMin,Ymax: Real);
begin
  CoordX := XMin + (XMax-XMin) * Random;
  CoordY := YMin + (YMax-YMin) * Random;
  Id := InitId;
end;

Procedure TCity.Ler (Var Arq : TextFile);
```

```
begin
  Readln (Arq,Id,CoordX,CoordY);
end;

Destructor TCity.Done;
begin
end;

Function TCity.PontoTela (UmPaintBox : TPaintBox) : TPoint;
begin
  Result.x := Round((CoordX  - XMin) * Escala) + 10;
  Result.y := UmPaintBox.Height - Round((CoordY - YMin) * Escala) - 10;
end;

Procedure TCity.Mostrar (UmPaintBox : TPaintBox; ComID : Boolean);
Var
  UmPto : TPoint;
begin
  With UmPaintBox.Canvas do
  begin
    UmPto := PontoTela (UmPaintBox);
    Pen.Color := clBlack;
    Rectangle (UmPto.X-1,UmPto.Y-1,UmPto.X+1,UmPto.Y+1);
    Font.Size:=5;
    If ComID then
    begin
      Brush.Style :=bsClear;
      TextOut (UmPto.X+5,UmPto.Y+5,IntToStr(Id));
    end;
  end;
end;

Procedure TCity.Gravar (Var Arq : TextFile);
begin
  Writeln (Arq,Id,CoordX,CoordY);
end;

end.
```

unit UArvMin;

```
interface
uses Graphics,Extctrls,Windows,Classes,Stdctrls,SysUtils,UCityList,Comctrls,forms;
Type
  TArray_of_Array_of_Int64 = Array of Array of Int64;
  TArco = record
          Ativo          : Boolean;
          ia,ic,jb,jd    : Integer;
          Comprimento    : Int64;
         end;
  PTVertice = ^TVertice;
  TVertice = object (TCityList)
               CoordX,CoordY   : Real;   {Coordenadas do Centro de Garvidade do Conjunto}
               ListaDeVizinhos : Array of Integer;
               Destructor ListaVizinhosDone;
             end;
  PTGrafo = ^TGrafo;
  TGrafo = object
   Public
    TotCidNosVertices      : Integer;
    MenorVertice,
    MaiorVertice           : Integer;
    Vertices               : TList;
    Arcos                  : Array of Array of TArco;
    ArcosArvMin            : Array of Array of Boolean;
    CustoArvMin            : Int64;
    RotaGlobal             : PTCityList;
    Constructor Init;
    Destructor Done;
    Destructor LimpaMemoria;
    Procedure InitArcos(QtdVert : Integer);
    Procedure InitArcosArvMin(QtdVert : Integer);
```

```
    Procedure LerCityList (NomeArquivo    : String;
                           MaxCidPorClus  : Integer;
                           MinCidPorClus  : Integer;
                           edMenorMaior   : TEdit;
                           UmTipoDist     : Integer;
                           UmPaintBox     : TPaintBox;
                           UmMemo         : TMemo);
    Procedure LerCityListMed (NomeArquivo   : String;
                              MaxCidPorClus : Integer;
                              UmTipoDist    : Integer;
                              UmPaintBox    : TPaintBox;
                              UmMemo        : TMemo );
    Procedure SetArco(i,j,a,c,b,d:Integer; TamArco : Int64);
    Procedure SetArcoArvMin(i,j : Integer);
    Function GetComprArco (i,j : Integer) : Int64;
    Function GetArcoArvMin (i,j : Integer) : Boolean;
    Function GetCustoArvMin : Int64;
    Function GetPTVertice (i: integer) : PTVertice;
    Function GetVertice_CoordX(i : integer) : Real;
    Function GetVertice_CoordY(i : integer) : Real;
    Function CalDisEntVert (i,j,UmTipoDist : Integer) : Int64;
    Function CalDisEntUmVerteUmPonto(i,UmTipoDist : integer; x,y:Real):Int64;
    Function AvaliaRotaGlobal : Int64;
    Procedure RealocaVertIsolados(IndVert,UmTipoDist : Integer);
    Procedure CriaListaDeVerticesVizinhos(k:integer; UmTipoDist:Integer);
    Procedure ConstroiArcos(UmMemo:TMemo; UmTipoDist  : Integer);
    Function CalcDistEntPontEmVertDifer(V1,V2,p1,p2,UmTipoDist:Integer):Int64;
    Procedure DetermCitiesMaisProxEntreVert( V1,V2               : Integer;
                                    Var UmaCity1,UmaCity2 : Integer;
                                    UmTipoDist            : Integer;
                                    Memo                  : TMemo );
    Procedure DetermConexMaisEconomicas(V1,V2       : Integer;
                                UmTipoDist  : Integer;
                                UmMemo      : TMemo);
    Procedure GeraRotaUnica(UmMemo:TMemo; UmPaintBox:TPaintBox);
    Procedure ListaArvMin(Memo : TMemo);
    Function EhSucessor(i,j : Integer) : boolean;
    Function EhSucessorArvMin(i,j : Integer) : boolean;
    Procedure GerarArvMin;
    Procedure CalcularCusto;
    Procedure MostraArvore (UmPaintBox : TPaintBox; ComID  : Boolean);
    Procedure OtimizaRotas (edParametro,edAproximacao,
                            edMaxIter,edIter,edOtimo,edCPU,
                            edNumTotClus,edNumClusResol,
                            edClusID,edNumCidClus      : TEdit;
                            TipoDist                   : Integer;
                            PaintBox                   : TPaintBox;
                            UmMemo                     : TMemo;
                            NVizinhos                  : Integer;
                            CalcDistDinam              : Boolean;
                            Variantes                  : Integer;
                            AutoAjuste                 : Boolean);
    Function SubdivideVertice(IndVert : Integer; MaxCidPorClus : Integer;
                              UmMemo : TMemo) : Boolean;
    Procedure MostraCG(VertID : Integer; UmVertice : PTVertice;
                              UmPaintBox : TPaintBox; ComID : Boolean);
    Procedure AnexaVertice(V1,V2      : Integer;
                           UmMemo     : TMemo;
                           UmPaintBox : TPaintBox);
  end;
Var
  UmGrafo : PTGrafo;
Procedure OrdVetInt64(Var A:TArray_of_Array_of_Int64; iLo,iHi :Integer);
Procedure OrdVetReal_B(Var A:TArray_of_Array_of_Real; Col, iLo,iHi :Integer);

implementation

Destructor TVertice.ListaVizinhosDone;
begin
  Finalize(ListaDeVizinhos);
end;

Destructor TGrafo.LimpaMemoria;
var
```

```
  i:integer;
  UmVertice :PTVertice;
begin
  For i:=0 to Vertices.Count-1 do
  begin
    UmVertice:=Vertices.Items[i];
    UmVertice.ListaVizinhosDone;
  end;
  Finalize(Arcos);
  Finalize(ArcosArvMin);
end;

Constructor TGrafo.Init;
begin
  CustoArvMin := 0;
  Vertices.Free;
  Vertices:=TList.Create;
end;

Procedure TGrafo.InitArcos(QtdVert : Integer);
Var
  i : Integer;
begin
   Arcos := nil;
   SetLength(Arcos,QtdVert);
   For i := Low(Arcos) to High(Arcos) do
     SetLength(Arcos[i],i+1);
end;

Procedure TGrafo.InitArcosArvMin(QtdVert : Integer);
Var
  i : Integer;
begin
   ArcosArvMin := nil;
   SetLength(ArcosArvMin,QtdVert);
   For i := Low(ArcosArvMin) to High(ArcosArvMin) do
     SetLength(ArcosArvMin[i],i+1);
end;

Procedure TGrafo.LerCityList (NomeArquivo    : String;
                              MaxCidPorClus  : Integer;
                              MinCidPorClus  : Integer;
                              edMenorMaior   : TEdit;
                              UmTipoDist     : Integer;
                              UmPaintBox     : TPaintBox;
                              UmMemo         : TMemo);
Var
  Arq               : TextFile;
  i,j,N             : Integer;
  X,Y               : Real;
  UmVertice         : PTVertice;
  UmaCidade         : PTCity;
  MedCoordX,MedCoordY : Real;
  Aux               : PTCity;
  Limite,
  MenorCluster,
  MaiorCluster      : Integer;
begin
  MenorCluster:=High(Integer);
  MaiorCluster:=Low(integer);
  XMin := 1E+30;
  YMin := 1E+30;
  XMax := -1E+30;
  YMax := -1E+30;
  AssignFile (Arq,NomeArquivo);
  Reset (Arq);
  New(UmVertice);
  Vertices.Add(UmVertice);
  UmVertice.ListCity := TList.Create;
  N:=0;
  While not EOF (arq) do
  begin
    N:=N+1;
    Readln (Arq,i,X,Y);
```

```
    If X < XMin then XMin := X;
    If Y < YMin then YMin := Y;
    If X > XMax then XMax := X;
    If Y > YMax then YMax := Y;
    New(UmaCidade,Init(i,X,Y));
    UmVertice.ListCity.Add(UmaCidade);
   end;
  TotCidNosVertices:=N;
  CloseFile (Arq);
  i:=0;
  Repeat
    If SubDivideVertice(i,MaxCidPorClus,UmMemo)=False then i:=i+1;
  Until i>=Vertices.Count;
  For i := 0 to Vertices.Count-1 do
  begin
    UmVertice:=Vertices.Items[i];
    MedCoordX:=0;
    MedCoordY:=0;
    For j:=0 to UmVertice.ListCity.Count-1 do
    begin
      Aux:=UmVertice.ListCity.Items[j];
      MedCoordX:=MedCoordX+Aux.CoordX;
      MedCoordY:=MedCoordY+Aux.CoordY;
    end;
    MedCoordX:=MedCoordX/UmVertice.ListCity.Count;
    MedCoordY:=MedCoordY/UmVertice.ListCity.Count;
    UmVertice.CoordX:=MedCoordX;
    UmVertice.CoordY:=MedCoordY;
  end;
  {Realoca cidades de vértices com 1 ou 2 cidades para os vertices mais próximos.
   A distância entre uma cidade e um vértice é dada pela distância entre cidade
   e centro de gravidade do vérice}
  i:=0;
  Limite:=MinCidPorClus;
  if Limite<3 then Limite:=3;
  Repeat
    UmVertice:=Vertices.items[i];
    if UmVertice.ListCity.count<Limite
    then RealocaVertIsolados(i,UmTipoDist)
    else i:=i+1;
  Until i>Vertices.Count-1;
  {Elimina os vértices nulos}
  Vertices.Pack;
  Vertices.Capacity:=Vertices.Count;
  For i:=0 to Vertices.Count-1 do
  begin
    UmVertice:=Vertices.Items[i];
    If UmVertice.ListCity.Count>MaiorCluster then MaiorCluster:=UmVertice.ListCity.Count;
    If UmVertice.ListCity.Count<MenorCluster then MenorCluster:=UmVertice.ListCity.Count;
    New (UmVertice.RotaCorrente,Init (clSilver,1));
    New (UmVertice.RotaOtima,Init (clBlue,1));
    UmVertice.RotaCorrente.GerarRandom(UmVertice.ListCity.Count);
    UmVertice.RotaOtima.Recebe (UmVertice.RotaCorrente);
  end;
  MenorVertice:=MenorCluster;
  MaiorVertice:=MaiorCluster;
  edMenorMaior.text:=inttostr(MenorVertice)+'/'+inttostr(MaiorVertice);
  UmPaintBox.Refresh;
end;

Function TGrafo.SubdivideVertice(IndVert :Integer; MaxCidPorClus:Integer;UmMemo:TMemo):Boolean;
Var
  UmVerticeBase,UmVertice : PTVertice;
  UmaCidadeBase,UmaCidade : PTCity;
  Nx,Ny,k,i,j             : Integer;
  XMin,XMax,YMin,YMax,
  dx,dy,dBasico           : Real;

begin
   Result:=False;
   UmVerticeBase:=Vertices.Items[IndVert];
   If UmVerticeBase.ListCity.Count=0
   then begin
        Vertices.Delete(IndVert);
```

```
        Vertices.Pack;
        Vertices.Capacity:=Vertices.Count;
        Result:=True;
       end
  else begin
        If UmVerticeBase.ListCity.Count>MaxCidPorClus then
        begin
          XMin := 1E30;
          YMin := 1E30;
          XMax := -1E30;
          YMax := -1E30;
          For k:= 0 to UmVerticeBase.ListCity.Count-1 do
          begin
            UmaCidadeBase:=UmVerticeBase.ListCity.Items[k];
            If UmaCidadeBase.CoordX < XMin then XMin := UmaCidadeBase.CoordX;
            If UmaCidadeBase.CoordY < YMin then YMin := UmaCidadeBase.CoordY;
            If UmaCidadeBase.CoordX > XMax then XMax := UmaCidadeBase.CoordX;
            If UmaCidadeBase.CoordY > YMax then YMax := UmaCidadeBase.CoordY;
          end;
          Nx:=1;
          Ny:=1;
          dBasico := sqrt((XMax-XMin) * (YMax-YMin) * MaxCidPorClus / UmVerticeBase.ListCity.Count);
          if (XMax=Xmin) and (Ymax=Ymin) then  UmMemo.lines.add('ABERRAção');
          if (XMax=Xmin) or (Ymax=Ymin)
          then begin
                if Xmax=Xmin
                then begin
                      Ny:=2;
                      Nx:=1;
                end;
                if Ymax=Ymin
                then begin
                      Ny:=1;
                      Nx:=2;
                    end;
                end
          else begin
                Nx := Trunc (((XMax - XMin) / dBasico)+1);
                Ny := Trunc (((YMax - YMin) / dBasico)+1);
               end;
          if (Nx=1) and (Ny=1)
          then if random>0.5
              then Nx:=2
              else Ny:=2;
          dX := (XMax - XMin) / Nx;
          dY := (YMax - YMin) / Ny;
          For k := 1 to Nx*Ny do
          begin
            New(UmVertice);
            Vertices.Insert(IndVert+k,UmVertice);
            UmVertice.ListCity:=TList.Create;
          end;
          For k := 0 to UmVerticeBase.ListCity.Count-1 do
          begin
            UmaCidadeBase:=UmVerticeBase.ListCity.Items[k];
            New (UmaCidade,Init (UmaCidadeBase.Id,UmaCidadeBase.CoordX,UmaCidadeBase.CoordY));
            j := Trunc ((UmaCidadeBase.CoordX-XMin) / dX);
            If j >= Nx then j := Nx - 1;
            i := Trunc ((UmaCidadeBase.CoordY-YMin) / dY);
            If i >= Ny then i := Ny - 1;
            UmVertice := Vertices.Items [IndVert + 1 + i*Nx + j];
            UmVertice.ListCity.Add (UmaCidade);
          end;
          Vertices.Delete(IndVert);
          Vertices.Pack;
          Vertices.Capacity:=Vertices.Count;
          Result:=True;
        end;
  end;
end;

Procedure TGrafo.MostraCG(VertID:Integer; UmVertice:PTVertice;UmPaintBox:TPaintBox; ComID:Boolean);
Var
  UmPto :TPoint;
```

```
begin
  If UmVertice<>nil
  then With UmPaintBox.Canvas do
       begin
         Pen.Color := clBlack;
         Brush.Style := bsSolid;
         Brush.Color := clRed;
         UmPto.x:=Round((UmVertice.CoordX  - XMin) * Escala) + 10;
         UmPto.y:= UmPaintBox.Height - Round((UmVertice.CoordY - YMin) * Escala) - 10;
         Ellipse (UmPto.X-3,UmPto.Y-3,UmPto.X+3,UmPto.Y+3);
         TextOut (UmPto.X+3,UmPto.Y+3,IntToStr(UmVertice.ListCity.Count));
         If ComID then
         begin
           Brush.Style := bsClear;
           TextOut (UmPto.X+3,UmPto.Y+3,IntToStr(VertID));
         end;
       end;
end;

Procedure TGrafo.LerCityListMed (NomeArquivo   : String;
                                 MaxCidPorClus : Integer;
                                 UmTipoDist    : Integer;
                                 UmPaintBox    : TPaintBox;
                                 UmMemo        : TMemo );
Var
  Arq            : TextFile;
  N,Nx,Ny,auxi,
  i,j,k,p        : Integer;
  X,Y,dBasico,
  dX,dY          : Real;
  UmVertice      : PTVertice;
  UmaCidade      : PTCity;
  MedCoordX,
  MedCoordY      : Real;
  Aux            : PTCity;
  VerticesAux    : TArray_of_Array_of_Real;
  a,b            : real;
  c,liminf,limsup : integer;
begin
  N := 0;
  AssignFile (Arq,NomeArquivo);
  Reset (Arq);
  XMin := 1E30;
  YMin := 1E30;
  XMax := -1E30;
  YMax := -1E30;
  While not EOF (arq) do
  begin
    N := N + 1;
    Readln (Arq,i,X,Y);
    If X < XMin then XMin := X;
    If Y < YMin then YMin := Y;
    If X > XMax then XMax := X;
    If Y > YMax then YMax := Y;
  end;
  TotCidNosVertices:=N;
  Reset (Arq);
  dBasico := sqrt((XMax-XMin) * (YMax-YMin) * MaxCidPorClus / N);
  Nx := Round ((XMax - XMin) / dBasico);
  Ny := Round ((YMax - YMin) / dBasico);
  Vertices:=Tlist.Create;
  For i := 0 to (Nx*Ny)-1 do
  begin
    New (UmVertice);
    Vertices.Add(UmVertice);
    UmVertice.ListCity := TList.Create;
    New (UmVertice.RotaCorrente,Init (clSilver,1));
    New (UmVertice.RotaOtima,Init (clRed,1));
  end;
  Setlength(VerticesAux,N,3);
  For k:=0 to N-1 do
  begin
    Readln (Arq,i,X,Y);
    VerticesAux[k,0]:=i;
```

```
    VerticesAux[k,1]:=X;
    VerticesAux[k,2]:=Y;
  end;
  OrdVetReal_B(VerticesAux,1,0,N-1);
//  Ajuste do eixo X
  For P:=1 to Nx do
  begin
   if p=1 then liminf:=1 else liminf:=((N*(P-1)) div Nx)+1;
   if P=Nx then limsup:=N else limsup:=(N*P) div Nx;
   For k:=liminf to limsup do
   begin
    New (UmaCidade,Init (trunc(VerticesAux[k-1,0]),VerticesAux[k-1,1],VerticesAux[k-1,2]));
    UmVertice := Vertices.Items [p-1];
    UmVertice.ListCity.Add (UmaCidade);
   end;
  end;
//  Ajuste do eixo Y
  For i:=1 to Nx do
  begin
   VerticesAux:=nil;
   UmVertice := Vertices[i-1];
   N:=UmVertice.ListCity.Count;
   Setlength(VerticesAux,N,3);
   For k:=0 to N-1 do
   begin
     UmaCidade:=UmVertice.ListCity.Items[k];
     VerticesAux[k,0]:=UmaCidade.Id;
     VerticesAux[k,1]:=UmaCidade.CoordX;
     VerticesAux[k,2]:=UmaCidade.CoordY;
   end;
   OrdVetReal_B(VerticesAux,2,0,N-1);
   UmVertice.ListCity.Destroy;
   UmVertice.ListCity := TList.Create;
   New (UmVertice.RotaCorrente,Init (clSilver,1));
   New (UmVertice.RotaOtima,Init (clRed,1));
   For p:=1 to Ny do
   begin
     if p=1 then liminf:=1 else liminf:=((N*(p-1)) div Ny)+1;
     if P=Ny then limsup:=N else limsup:=(N*p) div Ny;
     For k:=liminf to limsup do
     begin
       New (UmaCidade,Init (trunc(VerticesAux[k-1,0]),VerticesAux[k-1,1],VerticesAux[k-1,2]));
       UmVertice := Vertices.items [(p-1)*Nx+(i-1)];
       UmVertice.ListCity.Add (UmaCidade);
     end;
   end;
  end;
  For i := 0 to (Nx*Ny)-1 do
  begin
    MedCoordX:=0;
    MedCoordY:=0;
    UmVertice:=Vertices.Items[i];
    For j:=0 to UmVertice.ListCity.Count-1 do
    begin
      Aux:=UmVertice.LisTCity.items[j];
      MedCoordX:=MedCoordX+Aux^.CoordX;
      MedCoordY:=MedCoordY+Aux^.CoordY;
    end;
    MedCoordX:=MedCoordX/UmVertice.ListCity.Count;
    MedCoordY:=MedCoordY/UmVertice.ListCity.Count;
    UmVertice.CoordX:=MedCoordX;
    UmVertice.CoordY:=MedCoordY;
  end;
  CloseFile (Arq);
    UmPaintBox.Refresh;
end;

Procedure TGrafo.MostraArvore (UmPaintBox : TPaintBox; ComID : Boolean);
Var
  UmVertice : PTVertice;
  i         : Integer;
begin
  For i := 0 to Vertices.Count-1 do
  begin
```

```
    UmVertice:=Vertices[i];
    UmVertice.MostrarCity(UmPaintBox,ComID);
    UmVertice.MostrarRota(UmVertice.RotaOtima,UmPaintBox);
    MostraCG(i,UmVertice,UmPaintBox,ComID);
  end;
end;

Destructor TGrafo.Done;
begin
end;

Procedure TGrafo.SetArco(i,j,a,c,b,d:Integer; TamArco : Int64);
begin
  if (i>=j)
  then begin
         Arcos[i,j].Ativo := True;
         Arcos[i,j].ia:=a;
         Arcos[i,j].ic:=c;
         Arcos[i,j].jb:=b;
         Arcos[i,j].jd:=d;
         Arcos[i,j].Comprimento := TamArco;
       end
  else begin
         Arcos[j,i].Ativo := True;
         Arcos[j,i].ia:=c;
         Arcos[j,i].ic:=a;
         Arcos[j,i].jb:=d;
         Arcos[j,i].jd:=b;
         Arcos[j,i].Comprimento := TamArco;
       end;
end;

Procedure TGrafo.SetArcoArvMin(i,j : Integer);
begin
  if (i>=j) then ArcosArvMin[i,j] := True
            else ArcosArvMin[j,i] := True;
end;

Function TGrafo.GetComprArco(i,j : Integer) : Int64;
begin
  if (i>=j)
  then Result := Arcos[i,j].Comprimento
  else Result := Arcos[j,i].Comprimento;
end;

Function TGrafo.GetArcoArvMin (i,j : Integer) : Boolean;
begin
  if (i>=j)
  then Result := ArcosArvMin[i,j]
  else Result := ArcosArvMIn[j,i];
end;

Function TGrafo.GetPTVertice (i: integer) : PTVertice;
begin
  if (Vertices.Count>0) and (i>=0) and (i<=Vertices.Count-1)
  then Result:=Vertices[i]
  else Result:=nil;
end;

Function TGrafo.GetVertice_CoordX(i : Integer) : Real;
Var
 UmVertice:PTVertice;
begin
  UmVertice:=Vertices.Items[i];
  Result:=UmVertice.CoordX;
end;

Function TGrafo.GetVertice_CoordY(i : Integer) : Real;
Var
 UmVertice:PTVertice;
begin
  UmVertice:=Vertices.Items[i];
  Result:=UmVertice.CoordY;
end;
```

```
Function TGrafo.CalDisEntVert(i,j,UmTipoDist:integer):Int64;
Var
  V1,V2 : PTVertice;
begin
  V1:=Vertices.items[i];
  V2:=Vertices.items[j];
  Case UmTipoDist of
    0 : Result:= DistEuclid_2d(V1.CoordX,V1.CoordY,V2.CoordX,V2.CoordY);
    1 : Result:= DistCeil_2d(V1.CoordX,V1.CoordY,V2.CoordX,V2.CoordY);
  end;
end;

Function TGrafo.CalDisEntUmVerteUmPonto(i,UmTipoDist:integer; x,y:Real):Int64;
Var
  V1 : PTVertice;
begin
  V1:=Vertices.items[i];
  Case UmTipoDist of
    0 : Result:=DistEuclid_2d(V1.CoordX,V1.CoordY,x,y);
    1 : Result:=DistCeil_2d(V1.CoordX,V1.CoordY,x,y);
  end;
end;

Function TGrafo.GetCustoArvMin : Int64;
begin
   Result := CustoArvMin;
end;

Procedure TGrafo.GerarArvMin;
Type
  PTRotulo = ^TRotulo;
  TRotulo = Record
              Antec : Integer;
              Custo : Real;
            end;
Var
  i,UltVertEntList,
  Indice1,Indice2    : Integer;
  Min,Max            : Real;
  VertIn,VertOut     : Tlist;
  Aux                : ^Integer;
  Rotulos            : Array of TRotulo;
begin
  Max:=1.7*10E308;
  VertIn := Tlist.Create;
  VertOut := Tlist.Create;
  InitArcosArvMin(Vertices.Count);
  UltVertEntList := 0;
  New(Aux);
  Aux^ := UltVertEntList;
  VertIn.Add(Aux);
  for i := 0 to Vertices.Count-1 do
  begin
    if (UltVertEntList<>i)  then
    begin
      New(Aux);
      Aux^ := i;
      VertOut.Add(Aux);
    end;
  end;
  SetLength(Rotulos,Vertices.Count);
  for i := 0 to Vertices.Count-1 do
  begin
    Rotulos[i].Antec := -1;
    Rotulos[i].Custo := Max;
  end;
  While VertIn.Count<Vertices.Count do
  begin
     // Atualizando rótulos
     for i := 0 to VertOut.Count-1 do
     begin
        Aux := VertOut.Items[i];
        if EhSucessor(Aux^,UltVertEntList)
```

```
      then if (GetComprArco(Aux^,UltVertEntList)<Rotulos[Aux^].Custo)
           then begin
                Rotulos[Aux^].Antec := UltVertEntList;
                Rotulos[Aux^].Custo := GetComprArco(Aux^,UltVertEntList);
               end;
    end;
    // Buscando a aresta de mínimo custo
    Indice1 := 0;
    Aux := VertOut.Items[Indice1];
    Min := Rotulos[Aux^].Custo;
    Indice2 := Aux^;
    for i:= 1 to VertOut.Count-1 do
    begin
     Aux := VertOut.Items[i];
     if Rotulos[Aux^].Custo<Min
     then begin
           Indice1 := i;
           Indice2 := Aux^;
           Min := Rotulos[Aux^].Custo;
          end;
    end;
    // Fazendo as atualizações
    SetArcoArvMin(indice2,Rotulos[Indice2].Antec);
    VertOut.Delete(Indice1);
    New(Aux);
    Aux^ := indice2;
    VertIn.Add(Aux);
    UltVertEntList := indice2;
  end;
  CalcularCusto;
  VertIn.Free;
  VertOut.Free;
end;

Procedure  TGrafo.CalcularCusto;
Var
  i,j   : Integer;
  Custo : Int64;
begin
  Custo := 0;
  For i := 0 to Vertices.Count-1 do
    For j := 0 to i do
      if (ArcosArvMin[i,j]=True) then Custo := Custo + GetComprArco(i,j);
  CustoArvMin:=Custo;
end;

Function TGrafo.EhSucessor(i,j : Integer) : boolean;
begin
 if (i=j) then Result := false
         else if i>j then Result := Arcos [i,j].Ativo
                  else Result := Arcos [j,i].Ativo;
end;

Function TGrafo.EhSucessorArvMin(i,j : Integer) : boolean;
begin
 if (i=j) then Result := false
         else if i>j then Result := ArcosArvMin [i,j]
                  else Result := ArcosArvMin [j,i];
end;

Procedure Tgrafo.RealocaVertIsolados(IndVert, UmTipoDist : Integer);
Var
  UmVertice,UmVerticeDest    : PTVertice;
  UmaCidade                  : PTCity;
  i,j,jDest                  : Integer;
  Aux,MinDist                : Int64;
begin
  UmVertice:=Vertices.items[IndVert];
  For i:=0 to UmVertice.ListCity.Count-1 do
  begin
    UmaCidade:=UmVertice.ListCity.items[i];
    MinDist:=High(Int64);
    jDest:=0;
    For j:=0 to Vertices.Count-1 do
```

```
    begin
      UmVerticeDest:=Vertices.items[j];
      If (j<>IndVert)and (UmVerticeDest<>nil)
      then begin
                Aux:=CalDisEntUmVerteUmPonto(j,UmTipoDist,UmaCidade.CoordX,
                                             UmaCidade.CoordY);
                if Aux<MinDist
                then begin
                     MinDist :=Aux;
                     JDest   :=j;
                    end;
           end;
    end;
    UmVerticeDest:=Vertices.items[jDest];
    UmVerticeDest.ListCity.Add(UmaCidade);
  end;
  Vertices.Delete(IndVert);
  Vertices.Pack;
  Vertices.Capacity:=Vertices.Count;
end;

Procedure TGrafo.CriaListaDeVerticesVizinhos(k:integer; UmTipoDist:Integer);
Var
   i,j,t       : Integer;
   DistEntreCG : TArray_of_Array_of_Int64;
   UmVertice   : PTVertice;
begin
   if k>=Vertices.Count then k:=Vertices.Count-1;
   for i:=0 to Vertices.Count-1 do
   begin
     DistEntreCG:=nil;
     SetLength(DistEntreCG,Vertices.Count-1,2);
     t:=0;
     for j:=0 to Vertices.Count-1 do
       if i<>j
       then begin
            DistEntreCG[t,0]:= j;
            DistEntreCG[t,1]:= CalDisEntVert(i,j,UmTipoDist);
            t:=t+1;
           end;
     OrdVetInt64(DistEntreCG,0,Vertices.Count-2);
     UmVertice:=Vertices.items[i];
     SetLength(UmVertice.ListaDeVizinhos,k);
     for t:= 0 to k-1 do
     begin
       UmVertice.ListaDeVizinhos[t]:=DistEntreCG[t,0];
     end;
   end;
end;

Procedure TGrafo.ConstroiArcos(UmMemo:TMEmo; UmTipoDist  : Integer);
Var
   i,j                 : Integer;
   UmVertice1          : PTVertice;
begin
  InitArcos(Vertices.Count);
  for i:=0 to Vertices.Count-1 do
  begin
    UmVertice1:=Vertices.Items[i];
    for j:=Low(UmVertice1.ListaDeVizinhos) to High(UmVertice1.ListaDeVizinhos) do
      If EhSucessor(i,UmVertice1.ListaDeVizinhos[j])=False
      then DetermConexMaisEconomicas(i,UmVertice1.ListaDeVizinhos[j],UmTipoDist,UmMemo);
  end;
  GerarArvMin;
end;

Function TGrafo.CalcDistEntPontEmVertDifer(V1,V2,p1,p2,UmTipoDist:Integer):Int64;
var
  UmaCidade1,UmaCidade2 : PTCity;
  UmVertice1,UmVertice2 : PTVertice;
begin
  UmVertice1:=Vertices.Items[V1];
  UmaCidade1:=UmVertice1.ListCity.Items[p1];
  UmVertice2:=Vertices.Items[V2];
```

```
  UmaCidade2:=UmVertice2.ListCity.Items[p2];
  Case UmTipoDist of
  0 :  Result:=DistEuclid_2d(UmaCidade1.CoordX,UmaCidade1.CoordY,UmaCidade2.CoordX,UmaCidade2.CoordY);
  1 :  Result:=DistCeil_2d(UmaCidade1.CoordX,UmaCidade1.CoordY,UmaCidade2.CoordX,UmaCidade2.CoordY);
  end;
end;

Procedure TGrafo.DetermCitiesMaisProxEntreVert( V1,V2 : Integer; Var UmaCity1,UmaCity2:Integer;
                                    UmTipoDist:Integer; Memo:TMemo );
Var
  i,j                : Integer;
  UmVertice1,UmVertice2 : PTVertice;
  DistMin,Aux          : Int64;
begin
  DistMin:=High(DistMin);
  UmVertice1:=Vertices.items[V1];
  UmVertice2:=Vertices.Items[V2];
  For i:=0 to UmVertice1.ListCity.Count-1 do
  begin
    For j:=0 to UmVertice2.ListCity.Count-1 do
    begin
      Aux:=CalcDistEntPontEmVertDifer(V1,V2,i,j,UmTipoDist);
      If Aux<DistMin
      then begin
            DistMin:=Aux;
            UmaCity1:=i;
            UmaCity2:=j;
          end;
    end;
  end;
end;

Procedure TGrafo.DetermConexMaisEconomicas(V1,V2       : Integer;
                                  UmTipoDist  : Integer;
                                  UmMemo      : TMemo);
var
  i,City1,City2                : Integer;
  UmVertice1,UmVertice2            : PTVertice;
  UmaCity1,UmaAntCity1,UmaSucCity1,
  UmaCity2,UmaAntCity2,UmaSucCity2   : PTCity;
  PosCity1,AntCity1,SucCity1,
  PosCity2,AntCity2,SucCity2        : Integer;
  IndDMin                       : Integer;
  Dmin                          : Int64;
  D                            : Array [1..6] of Int64;

begin
  City1:=0;
  City2:=0;
  DetermCitiesMaisProxEntreVert(V1,V2,City1,City2,UmTipoDist,UmMemo);
  UmVertice1:=Vertices.Items[V1];
  UmVertice2:=Vertices.Items[V2];

  UmaCity1:=UmVertice1.ListCity.Items[City1];
  PosCity1:=UmVertice1.RotaOtima.GetPos(City1);

  AntCity1:=UmVertice1.RotaOtima.GetSuc(PosCity1,-1);
  UmaAntCity1:=UmVertice1.ListCity.Items[AntCity1];
  SucCity1:=UmVertice1.RotaOtima.GetSuc(PosCity1,1);
  UmaSucCity1:=UmVertice1.ListCity.Items[SucCity1];

  UmaCity2:=UmVertice2.ListCity.Items[City2];
  PosCity2:=UmVertice2.RotaOtima.GetPos(City2);

  AntCity2:=UmVertice2.RotaOtima.GetSuc(PosCity2,-1);
  UmaAntCity2:=UmVertice2.ListCity.Items[AntCity2];
  SucCity2:=UmVertice2.RotaOtima.GetSuc(PosCity2,1);
  UmaSucCity2:=UmVertice2.ListCity.Items[SucCity2];

  D[1]:= CalcDistEntPontEmVertDifer(V1,V2,AntCity1,AntCity2,UmTipoDist)+
        CalcDistEntPontEmVertDifer(V1,V2,City1,City2,UmTipoDist)-
        CalcDistEntPontEmVertDifer(V1,V1,AntCity1,City1,UmTipoDist)-
        CalcDistEntPontEmVertDifer(V2,V2,AntCity2,City2,UmTipoDist);
```

```
  D[2]:= CalcDistEntPontEmVertDifer(V1,V2,SucCity1,SucCity2,UmTipoDist)+
         CalcDistEntPontEmVertDifer(V1,V2,City1,City2,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V1,V1,SucCity1,City1,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V2,V2,SucCity2,City2,UmTipoDist);

  D[3]:= CalcDistEntPontEmVertDifer(V1,V2,AntCity1,SucCity2,UmTipoDist)+
         CalcDistEntPontEmVertDifer(V1,V2,City1,City2,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V1,V1,AntCity1,City1,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V2,V2,SucCity2,City2,UmTipoDist);

  D[4]:= CalcDistEntPontEmVertDifer(V1,V2,SucCity1,AntCity2,UmTipoDist)+
         CalcDistEntPontEmVertDifer(V1,V2,City1,City2,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V1,V1,SucCity1,City1,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V2,V2,AntCity2,City2,UmTipoDist);

  D[5]:= CalcDistEntPontEmVertDifer(V1,V2,AntCity1,City2,UmTipoDist)+
         CalcDistEntPontEmVertDifer(V1,V2,City1,SucCity2,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V1,V1,AntCity1,City1,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V2,V2,City2,SucCity2,UmTipoDist);

  D[6]:= CalcDistEntPontEmVertDifer(V1,V2,SucCity1,City2,UmTipoDist)+
         CalcDistEntPontEmVertDifer(V1,V2,City1,AntCity2,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V1,V1,SucCity1,City1,UmTipoDist)-
         CalcDistEntPontEmVertDifer(V2,V2,City2,AntCity2,UmTipoDist);

  DMin:=D[1];
  IndDMin:=1;
  For i:=2 to 6 do
   if D[i]<Dmin then begin
                      Dmin:=D[i];
                      IndDmin:=i;
                     end;
  Case IndDmin of
    1:   SetArco(V1,V2,UmaAntCity1.Id,UmaAntCity2.Id,UmaCity1.Id,UmaCity2.Id,DMin);
    2:   SetArco(V1,V2,UmaSucCity1.Id,UmaSucCity2.Id,UmaCity1.Id,UmaCity2.Id,DMin);
    3:   SetArco(V1,V2,UmaAntCity1.Id,UmaSucCity2.Id,UmaCity1.Id,UmaCity2.Id,DMin);
    4:   SetArco(V1,V2,UmaSucCity1.Id,UmaAntCity2.Id,UmaCity1.Id,UmaCity2.Id,DMin);
    5:   SetArco(V1,V2,UmaAntCity1.Id,UmaCity2.Id,UmaCity1.Id,UmaSucCity2.Id,DMin);
    6:   SetArco(V1,V2,UmaSucCity1.Id,UmaCity2.Id,UmaCity1.Id,UmaAntCity2.Id,DMin);
  end;
end;

Procedure TGrafo.AnexaVertice(V1,V2:Integer;UmMemo:TMemo; UmPaintBox:TPaintBox);
Var
  UmVertice                      : PTVertice;
  UmaCity,UmaCityAux,
  CityA,CityC,
  SucCityA_List,
  SucCityC_Rota,
  AntCityA_Rota                    : PTCity;
  UmaRota                        : PTRota;
  i,j,k,t,A,B,C,D,Spin,
  PosCityA_List,PosSucCityA_List,
  PosCityC_List,PosCityC_Rota,
  PosAntCityC_Rota,PosSucCityC_Rota,
  PosEntrada,PosInicialV2            : Integer;
begin
  if v1>=v2
  then begin
         A:=Arcos[v1,v2].ia;
         C:=Arcos[v1,v2].ic;
         B:=Arcos[v1,v2].jb;
         D:=Arcos[v1,v2].jd;
       end
  else begin
         A:=Arcos[v2,v1].ic;
         C:=Arcos[v2,v1].ia;
         B:=Arcos[v2,v1].jd;
         D:=Arcos[v2,v1].jb;
       end;
  UmVertice:=Vertices.Items[v2];
  UmaRota:=UmVertice.RotaOtima;
  {Localiza cidade "A" ,na rota global, e é o pivô na conexão}
  PosCityA_List:=-1;
```

```
  Repeat
    PosCityA_List:=PosCityA_List+1;
    CityA:=RotaGlobal.ListCity.Items[PosCityA_List];
  Until CityA.ID=A;
  {Identifica a cidade SUCESSORA da cidade A na lista de cidades da
   rota global.Retorna o ponteiro para a cidade e a posição do
   ponteiro na lista.}
  If PosCityA_List=RotaGlobal.ListCity.Count-1
     then PosSucCityA_List:=0
     else PosSucCityA_List:=PosCityA_List+1;
  SucCityA_List:=RotaGlobal.ListCity.items[PosSucCityA_List];
  {Localiza na lista de cidades do vértice que está entrando,
   a cidade "C" que conecta-se com a cidade "A" que já está na rota global}
  PosCityC_List:=-1;
  Repeat
      PosCityC_List:=PosCityC_List+1;
      CityC:=UmVertice.ListCity.Items[PosCityC_List];
  Until CityC.Id=C;
  PosCityC_Rota:=UmaRota.GetPos(PosCityC_List);
  {Identifica a cidade SUCESSORA da cidade C na rota do vertice entrando.
   Retorna o ponteiro para a cidade e a posição na rota}
  PosSucCityC_Rota:=UmaRota.GetPos(UmaRota.GetSuc(PosCityC_Rota,1));
  SucCityC_Rota:=UmVertice.ListCity.Items[UmaRota.GetNo(PosSucCityC_Rota)];
  PosAntCityC_Rota:=UmaRota.GetPos(UmaRota.GetSuc(PosCityC_Rota,-1));
  If  (SucCityA_List.ID=B) and (SucCityC_Rota.Id=D)
  then begin
         PosEntrada:=PosSucCityA_List;
         PosInicialV2:=PosSucCityC_Rota;
         Spin:=1;
       end;
  If  (SucCityA_List.ID=B) and (SucCityC_Rota.Id<>D)
  then begin
         PosEntrada:=PosSucCityA_List;
         PosInicialV2:=PosAntCityC_Rota;
         Spin:=-1;
       end;
  If  (SucCityA_List.ID<>B) and (SucCityC_Rota.Id=D)
  then begin
         PosEntrada:=PosCityA_List;
         PosInicialV2:=PosCityC_Rota;
         Spin:=-1;
       end;
  If  (SucCityA_List.ID<>B) and (SucCityC_Rota.Id<>D)
  then begin
         PosEntrada:=PosCityA_List;
         PosInicialV2:=PosCityC_Rota;
         Spin:=1;
       end;
  For t:=0 to UmVertice.ListCity.Count-1 do
  begin
    k:=UmaRota.GetSuc(PosInicialV2,spin*t);
    UmaCityAux:=UmVertice.ListCity.Items[k];
    New (UmaCity,Init(UmaCityAux.Id,UmaCityAux.CoordX,UmaCityAux.CoordY));
    RotaGlobal.ListCity.Insert(PosEntrada,UmaCity);
  end;
end;

Function Tgrafo.AvaliaRotaGlobal:Int64;
Var
  i : Integer;
  Cid1,Cid2 : PTCity;
begin
  Result := 0;
  If RotaGlobal.ListCity.Count<=1
  then Result:=0
  else begin
         For i := 0 to RotaGlobal.ListCity.Count - 2 do
         begin
           Cid1 := RotaGlobal.ListCity.items[i];
           Cid2 := RotaGlobal.ListCity.items[i+1];
           Result := Result + DistEuclid_2d(Cid1.CoordX,Cid1.CoordY,Cid2.CoordX,Cid2.CoordY);
         end;
         Cid1 := RotaGlobal.ListCity.items[0];
         Cid2 := RotaGlobal.ListCity.items[RotaGlobal.ListCity.Count - 1];
```

```
        Result:=Result+DistEuclid_2d(Cid1.CoordX,Cid1.CoordY,Cid2.CoordX,Cid2.CoordY);
      end;
end;

Procedure TGrafo.GeraRotaUnica(UmMemo:TMemo; UmPaintBox:TPaintBox);
var
  i,j                : integer;
  UmVertice          : PTVertice;
  UmaCity,UmaCityAux : PTCity;
  UmaRota            : PTRota;
  Sinal              : Array of Integer;
        Function TodosSinaisNeg(Var Sinal:Array of Integer):Boolean;
        Var
          k: integer;
        begin
          Result:=True;
          k:=0;
          While (k<=High(Sinal)) and (Result=True) do
          begin
            if Sinal[k]>-1 then Result:=False;
            k:=k+1;
          end;
        end;
Begin
  New(RotaGlobal);
  RotaGlobal.ListCity:= TList.Create;
  UmVertice:=Vertices.Items[0];
  UmaRota:=UmVertice.RotaOtima;
  For i:=0 to UmVertice.ListCity.Count-1 do
  begin
    j:=UmaRota.GetNo(i);
    UmaCityAux:=UmVertice.ListCity.Items[j];
    New (UmaCity,Init(UmaCityAux.Id,UmaCityAux.CoordX,UmaCityAux.CoordY));
    RotaGlobal.ListCity.Add(UmaCity);
  end;
  SetLength(Sinal,Vertices.Count);
  Sinal[0]:=1;
  While not(TodosSinaisNeg(Sinal)) do
  begin
    For i:=0 to Vertices.Count-1 do
    begin
      if Sinal[i]=1
      then begin
            For j:=0 to Vertices.Count-1 do
              if EhSucessorArvMin(i,j) and (Sinal[j]>-1)
              then begin
                    AnexaVertice(i,j,UmMemo,UmPaintBox);
                    Sinal[j]:=1;
                   end;
            Sinal[i]:=-1;
          end;
    end;
  end;
  New(RotaGlobal.RotaOtima,Init(clAqua,1));
  New(RotaGlobal.RotaCorrente,Init(clAqua,1));
  SetLength(RotaGlobal.RotaOtima.No,RotaGlobal.ListCity.Count);
  SetLength(RotaGlobal.RotaCorrente.No,RotaGlobal.ListCity.Count);
  For i:=0 to RotaGlobal.ListCity.Count-1 do
    RotaGlobal.RotaOtima.No[i]:=i;
  UmaRota:=RotaGlobal.RotaOtima;
  UmaRota.NRota:=RotaGlobal.ListCity.Count;
  UmaRota.Custo:=AvaliaRotaGlobal;
  RotaGlobal.RotaCorrente.Recebe((RotaGlobal.RotaOtima));
  LimpaMemoria;
end;

Procedure TGrafo.ListaArvMin(Memo : TMemo);
Var i,j:integer;
begin
  For i:=0 to Vertices.Count-1 do
    For j:=0 to i do
      if(ArcosArvMin[i,j]=true)then Memo.lines.add('('+inttostr(i)+','+inttostr(j)+')');
end;
```

```
Procedure TGrafo.OtimizaRotas (edParametro,edAproximacao,
                          edMaxIter,edIter,
                          edOtimo,edCPU,
                          edNumTotClus,edNumClusResol,
                          edClusID,edNumCidClus      : TEdit;
                          TipoDist                   : Integer;
                          PaintBox                   : TPaintBox;
                          UmMemo                     : TMemo;
                          NVizinhos                  : Integer;
                          CalcDistDinam              : Boolean;
                          Variantes                  : Integer;
                          AutoAjuste                 : Boolean);
Var
  UmVertice    : PTVertice;
  UmaRota      : PTRota;
  i            : Integer;
  CPU          : Real;
begin
  CPU := 0;
  edNumTotClus.text:=inttostr(Vertices.Count);
  For i := 0 to Vertices.Count - 1 do
  begin
    UmVertice := Vertices [i];
    edClusID.text:=inttostr(i);
    edNumCidClus.text:=Inttostr(UmVertice.ListCity.Count);
    If UmVertice.ListCity.Count > 0
    then begin
          UmaRota := UmVertice.RotaOtima;
          UmVertice.GlsTSP (edParametro,edAproximacao,edMaxIter,
                            edIter,edOtimo,edCPU,PaintBox,UmMemo,
                            UmaRota.Cor,UmaRota.LarguraLinha,
                            TipoDist,NVizinhos,Variantes,AutoAjuste);
          edNumClusResol.text:=inttostr(i+1);
          CPU := CPU + StrToFloat (edCPU.Text);
         end;
  end;
  Application.ProcessMessages;
end;

Procedure OrdVetInt64(Var A:TArray_of_Array_of_Int64; iLo,iHi :Integer);
Var
  Lo, Hi  : Integer;
  T, Mid  : Int64;
begin
  Lo := iLo;
  Hi := iHi;
  Mid := A[(Lo + Hi) div 2,1];
  repeat
    while A[Lo,1] < Mid do Inc(Lo);
    while A[Hi,1] > Mid do Dec(Hi);
    if Lo <= Hi
    then begin
          T := A[Lo,1];
          A[Lo,1] := A[Hi,1];
          A[Hi,1] := T;
          T := A[Lo,0];
          A[Lo,0] := A[Hi,0];
          A[Hi,0] := T;
          Inc(Lo);
          Dec(Hi);
         end;
  until Lo > Hi;
  if Hi > iLo then OrdVetInt64(A, iLo, Hi);
  if Lo < iHi then OrdVetInt64(A, Lo, iHi);
end;

Procedure OrdVetReal_B(Var A:TArray_of_Array_of_Real;Col,iLo,iHi :Integer);
Var
  Lo, Hi : Integer;
  T, Mid : Real;
begin
  Lo := iLo;
  Hi := iHi;
  Mid := A[(Lo + Hi) div 2,Col];
```

```
  repeat
    while A[Lo,Col] < Mid do Inc(Lo);
    while A[Hi,Col] > Mid do Dec(Hi);
    if Lo <= Hi
    then begin
           T := A[Lo,Col];
           A[Lo,Col] := A[Hi,Col];
           A[Hi,Col] := T;
           T := A[Lo,0];
           A[Lo,0] := A[Hi,0];
           A[Hi,0] := T;
           T := A[Lo,(Col mod 2)+1];
           A[Lo,(Col mod 2)+1] := A[Hi,(Col mod 2)+1];
           A[Hi,(Col mod 2)+1] := T;
           Inc(Lo);
           Dec(Hi);
         end;
  until Lo > Hi;
  if Hi > iLo then OrdVetReal_B(A, Col, iLo, Hi);
  if Lo < iHi then OrdVetReal_B(A, Col, Lo, iHi);
end;

end.
```